Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Roxana Ioana Roman

Community-based Recommendations to Improve Intranet Users' Productivity

Universe Community Recommender

Master's Thesis
Espoo, June 27, 2016

Supervisors: Associate Prof. Keijo Heljanko, Aalto University
Instructor: Perttu Ristimella M.Sc. (Tech.), Gapps

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Roxana Ioana Roman

Title:
Community-based Recommendations to Improve Intranet Users' Productivity
Universe Community Recommender

Date: June 27, 2016 **Pages:** 73

Supervisors: Associate Prof. Keijo Heljanko, Aalto University
Instructor: Perttu Ristimella M.Sc. (Tech.), Gapps

Intranet platforms are widely used in companies to facilitate the communication and collaboration between employees. Universe is an Intranet platform, integrating Google Apps such as Google Drive or Google Calendar, to allow groups of users to share documents and calendars. These collaborative tools can easily become overloaded with data and the employees no longer find relevant information in a reasonable amount of time. This thesis presents Universe Community Recommender, which augments Universe with a system that observes inter-user interactions and makes recommendations based on the community the user is part of. The system is composed of two main modules: Community Detector and Recommender Engine. The Community Detector implements two algorithms for detecting communities: Louvain and MCL. In case of Universe data, Louvain was chosen as the better choice due to the quality of communities detected. Based on the Universe users' graph, Louvain detects communities with a finer level of granularity than MCL. When comparing the execution time, MCL method tends to take a longer time when the number of users increase, because of its matrix multiplication procedure. Once the communities are detected, an HR administrator can visualize the communities, study the overall tendency in the company and detect the isolated users. The community structures also contribute at displaying more relevant information to the user. In our case, the user's main page will contain content posted by his community. Moreover, the user receives suggestions to join the channels his/her community is part of. Finally, the Universe users are presented with a version of Universe where these recommendations are employed and their feedback is gathered.

Keywords: Clustering Algorithms, Community Detection, Recommender Systems, Intranet, Google App Engine, Google Apps, Performance

Language: English

Acknowledgements

I want to thank Associate Professor Keijo Heljanko and my instructors Antero Hanhirova and Perttu Ristimella for their guidance.

Espoo, June 27, 2016

Roxana Ioana Roman

Abbreviations and Acronyms

UCR	Universe Community Recommender
CD	Community Detector
RE	Recommender Engine
DB	database
MCL	Markov Cluster Algorithm
$\mathbf{\Delta Q}$	modularity gain of moving node i to community C
$\sum_{\text{in}}$	sum of weights inside community C
$\sum_{\text{tot}}$	sum of weights of the edges incident to nodes inside community C
$\mathbf{k}_{\text{i}}$	sum of weights incident to node i
$\mathbf{k}_{\text{in}}$	sum of weights of the edges starting in node i to nodes belonging to community C
$\mathbf{m}$	total sum of weights in the network

Contents

Chapter 1

Introduction

Nowadays, the society is naturally divided into different categories: families, work colleagues, friends, etc. Nevertheless, these organizations can be obtained from the online communities as well. Moreover, communities can be found in all types of networked systems, such as protein to protein interactions, politics or computer networks. The first objective of this thesis is to study the existing community detection algorithms and apply them on an Intranet network.

In the context of today's amount of data, clustering and detecting communities is a necessity, as it brings many social and business benefits. For example, clustering Internet consumers with similar interests and geographically located in the same area, can improve the performance of the Internet services, each cluster being served by a dedicated server [11]. Moreover, in the e-commerce field, clustering users that like similar products is essential for providing targeted advertisements and avoiding spam. Clustering data regarding clients enables retailers like Amazon to build powerful recommender engines that eventually increase their sales. Google AdSense is another popular tool that tracks users' behaviour in order to display the more interesting advertisements on the right pages. Detecting communities can also improve search engines. Users with common interests are likely to input similar queries. Last but not least, graph clustering has an important role in computer networks and more specifically in routing. Routers are considered nodes in a graph. When building a routing table, knowing which nodes are central (i.e. are connected to a large number of other nodes) and which nodes are at the boundary between clusters will improve the routing mechanism and ease the network administration [7].

With the uncontrollable growth of the Internet, the users have identified a need for a technology to help them find what they need and filter what is relevant for them. A recommender engine builds the user profile, models social behaviour and brings interesting items closer to the user.

Last but not least and the purpose of this thesis, detecting communities inside a network can lead to making good recommendations to users regarding interesting content from their connections. For example, Facebook is selecting the posts that are showed first on your timeline depending on who posted them and recommends you contacts with similar interests. Furthermore, clustering helps detecting isolated users. This enables the social network company to identify and enhance these isolated users in the network and improve their experience. Other benefits of clustering in social networks can emerge depending on the network itself and its applications. Social networks produce a huge amount of data and choosing the most suitable graph clustering algorithm depends a lot on the structure of the network and the types of clusters needed to obtain.

This thesis will focus on finding the best approach to cluster users and make recommendations for **Universe**, an Intranet platform designed to ease the collaborative work between employees. The solution can be adapted to other Intranet solution especially the ones integrating Google Apps.

1.1 Problem statement

Online advertisement platforms such as Amazon, Google AdSense or Rakuten invest a lot of effort into reasoning about the user's preferences. This not only ends up by presenting the users with relevant content, but also increases the retailers' sales. In spite of the various benefits these recommendations have for the online advertisements field, the majority of the Intranet collaboration platforms do not offer them. The Intranet solutions do not reason about their users to simplify their experience. This leads to an Intranet platform where the useful content is lost through the irrelevant data, making it difficult for the employee to be productive and efficient. Being a collaborative tool for the working environment, Universe has to boost the user's productivity and make the communication and interaction easier inside companies. Hence, this thesis's aim is to augment Universe with a system that presents the user with relevant content at the right moment to speed up daily tasks.

1.2 Research scopes and goals

What makes a good online community in a social network? What is the proper algorithm for clustering data inside Universe? What recommendations can be done using community detection in Universe? These are the questions that this thesis aims at tackling.

1.2.1 Objectives

The main objectives of this project are to study, design and implement a recommendation system for Universe Intranet users. More exactly, the project is focusing on making these recommendations based on the communities formed inside Universe Intranet. The nodes in the graph are the users in Universe, while the edges are defined by the interactions between these users. The program must be able to respond and easily adapt to the multitude of actions that each user performs (e.g., creating new posts, commenting on other posts, editing Google Drive documents, etc). These happen every second, hence detecting the communities could be done as a batch job, every night, so that the communities adjust to the interactions during the day. A second option would be to compute the communities in real time. In this case, the user requests should not take longer than 1 second, which is a hard task to achieve for a community detection algorithm. Nevertheless, the performance will highly depend on the number of users clustered and the community detection algorithm chosen. The program should be easily extended and new algorithms for community detection effortlessly plugged into the project.

Firstly, user data has to be collected to create the vertices of the graph. Secondly, edge weights have to be computed. The weight depends on the number of common channels, the working department, the post/commenting activities and the Google Drive and Google Calendar interactions between the vertices making the edge. Next, having the graph structure of the company, a community detection algorithm is applied on the graph. Once the communities have been identified, some recommendations for the user have to be made. There are two types of recommendations:

1. *Based on the social graph:*
    (a) For each user display his/her first three most connected people.
    (b) Show a personalized My Stream, where the posts created by the user's community are displayed first.

   (c) Suggest channel creation for connected users.

2. *Based on the user's own activity:*

   (a) For each user, show the Drive Document the user has last seen or worked on.

   (a) For each user, display the meetings for the current day.

   (a) For each user, display the last unread emails.

Moreover, the social graph could be visualized by an HR person, who can observe how teams work together and who are the isolated users. All in a nutshell, the goals are:

1. *Construct a weighted matrix for all the users in the company.*
2. *Implement two community detection algorithms.*
3. *Compare the communities obtained with different algorithms.*
4. *Make the recommendations described before.*
5. *Display the social graph for the HR department in a user-friendly way.*

1.2.2 Functional requirements

The functional requirements of the Universe Community Recommender include:

1. *An API Endpoint that returns the community members of the logged in user.*
2. *An API Endpoint that returns the three most connected users to the logged in user.*
3. *An API Endpoint that returns a personalized My Stream.*
4. *An API Endpoint that returns user's My Workbench.*
5. *An API Endpoint for gathering user feedback regarding the recommendations made.*
6. *An API Endpoint for computing the communities inside the company.* The parameter to this call is the community detection algorithm. The output is a graph structure, where the connected components represent communities inside Universe.

7. *Identify, configure and compare scalable databases suitable for storing graph data.*

1.2.3 Non-functional requirements

One of the key non-functional requirements is **performance**. The endpoints that return results to the logged in user (from 1 to 5) should execute in less than one second. The endpoint for computing communities will take longer because it not only makes Google API calls to retrieve users' activities, but it also combines these results in order to create a weighted matrix. Therefore, it is the most appropriate to be run as a batch job.

Another key requirement is **extensibility**. New queries based on the social graph should be easily plugged in. Moreover, adding new clustering algorithms should not change the existing implementation and should offer the administrator the possibility to select between these algorithms when making the request.

Last but not least, **modularity** leads to a fast and easy to extend application. With Google App Engine, App Engine Modules can be used. These are logical components that can share resources and can communicate in a secure way. Hence, implementing Universe Community Recommender system in a separate Google App Engine module will contribute to the overall modularity of Universe.

1.3 Organization of the thesis

This thesis presents Universe Community Recommender, a complex solution that is able to detect communities inside a user network and to make recommendations to the user. This will lead to an improvement of the user experience and productivity on the Intranet platform. The thesis contributions include implementing and applying some of the existing clustering algorithms and recommendation techniques on a real social network. In Chapter 2, we discuss the existing solutions for the community detection problem. Then, we dive into the existing recommendation models and discuss how the information surrounding a user can improve the final recommendation. In Chapter 3, we present Universe, the Intranet solution for which Universe Community Recommender was implemented. Then, we analyse some of the existing tools for exploring and visualizing graphs. We study their usability in a production

environment and their potential to be integrated in the workflow of a social network application. Next, we analyse database systems for storing graph information and community structures and their potential to be integrated in the final workflow. In Chapter 4, we present the methods used and the overall system created for Universe. Chapters 5 and 6 present the details of the two components of the system: Community Detector and Recommender Engine. In Chapter 7, we present the manner in which the system has been evaluated by the users and the results obtained with different algorithms and tools. We finalize by exposing some limitations and present possible future improvements of the system in Chapter 8.

Our end goal is to augment Universe with a set of intelligent features that increase the user's productivity at work and enhance the user's experience. Universe Community Recommender would permit the platform to reason about its users. The community detection processes are scheduled tasks as they take longer to finish, while the recommendations are given in real time. The recommendations will ease the work of the employee and will offer a more personalized experience with Universe, while the social graph will provide the HR department with an overview over the interactions between employees inside the company, allowing it to form better teams and integrate new employees easier.

These features will be integrated in a separate Google App Engine Module, hence should not affect any current requests.

Chapter 2

Literature Review

This section presents an overview of the existing research and solutions in the field of graph clustering and recommendation systems. It tackles the most well-known social graph clustering algorithms and solutions. Last but not least, a series of Recommender Systems and the techniques applied are described.

2.1 Overview of community detection techniques

A community can be defined as a sub-graph or cluster of a graph. More precisely, it is as a subset of the graph's vertices, where the vertices that constitute the sub-graph have some predefined characteristics in common. The difference between graph clustering and community detection algorithms is, according to [23], the focus on different attributes when partitioning the graph. While community detection algorithms traditionally focus on the network structure, the clustering algorithms mostly consider only node attributes. However, the terms are usually interchanged and nowadays used as synonyms. The community detection problem has emerged with the increased popularity of social networks. Nonetheless, the concept behind it, which is graph clustering or graph partitioning, has been widely studied in graph theory and used in fields like molecular biology, network systems or others. The key idea is that in these systems data can be represented as graphs. Most often, the networks are weighted, that is the network has a scalar weight assigned to each of its edges. In Figure 2.1, a well-known

community example used as a benchmark for the clustering algorithms is displayed. It represents the structure of Zachary's network of karate club. It has 34 nodes, representing the members of the club. The edges represent the connections between the members outside the club activities. They were observed for a period of three years.

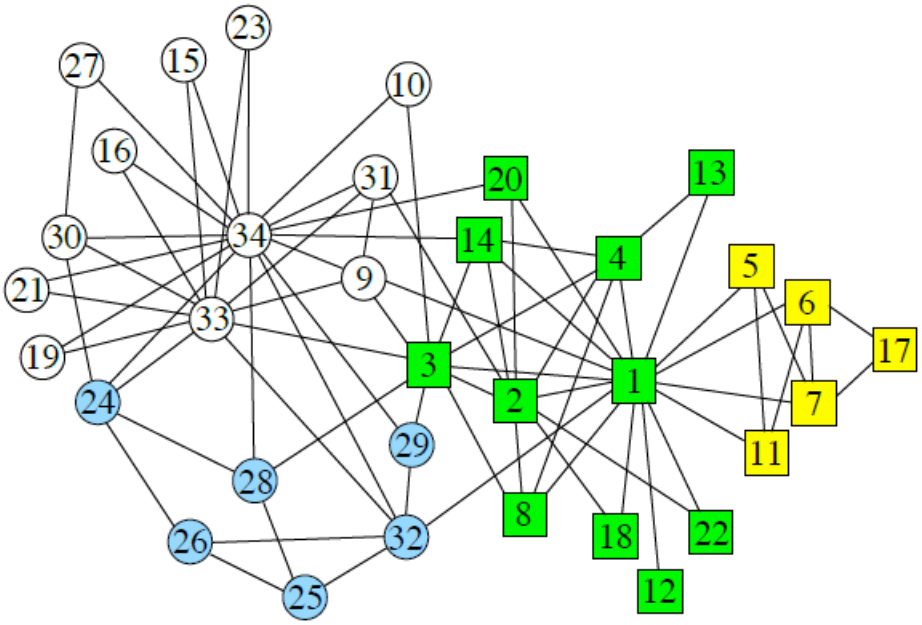

Figure 2.1: Zachary Karate Club (taken from [11])

2.1.1 Traditional clustering

The minimum cut method [6] is one of the oldest and most well-known techniques for graph partitioning. In graph theory, a cut separates a graph into two disjoint subsets of vertices. Depending on the network type, the minimum cut can be computed in different ways. For example, in an undirected graph, the minimum cut is the cut that contains the minimum number of edges. However, in a weighted graph, the minimum cut is defined as the cut that has the minimum weight among the all possible cuts. Most of the algorithms that compute the minimum cut perform a bisection of the graph, that is, the graph is partitioned in two groups. In order to obtain more groups an iterative bi-sectioning technique is applied on the initial graph. Therefore,

among the drawbacks of this method is that it can only find a fixed number of sub-graphs [8]. Moreover, iteratively partitioning the groups increases rapidly the run-time and storage space. Hence, the minimum cut is not suitable for finding communities in a general, large-sized network [15]. Another popular technique is the spectral bisection method, which was developed by Barnes in 1982 [2]. This is implemented using the Laplacian matrix. The main idea is that every partition of a graph can be represented by an index vector s, whose value s_i is 1 if i is a vertex belonging to the partition or -1 otherwise. The cut size will be computed using the transpose of the vector s and the Laplacian matrix, using the formula:

$$R = 1/4 \times s^T L_s, \tag{2.1}$$

where s^T is the transpose of vector s and L_s is the Laplacian matrix.

Random walks is another technique for detecting communities. The idea is that if a random walker spends a long time inside a cluster, it is due to the high number of connections between the nodes belonging to that cluster [15]. The random walks algorithms can be extended to weighted graphs as well [15].

Two other community detection techniques are clustering based on the shortest path betweenness and clustering based on the network modularity [8]. The basic steps include finding the edge that connects two communities, removing this edge and then checking if there is a disconnected component [8]. The disconnected component will correspond to a detected community. There are various ways to find these edges. One of them is edge betweenness, which implies finding the shortest paths among all nodes and checking which one goes through the edge that supposedly separates the cluster.

It is common to have a hierarchical structure inside a social graph and it is thus necessary to develop community detection techniques able to detect this type of structures. The hierarchical clustering technique consists in building a similarity matrix by taking every pair of vertices and computing the similarity between the two vertices of the pair [7]. The techniques for clustering aim at selecting the nodes with the highest similarity in the same cluster. There are two types of techniques in hierarchical clustering [7]:

1. *Agglomerative algorithms*, if the similarity between two clusters is high enough than these clusters are merged [7].
2. *Divisive algorithms*, if the similarity between two vertices is low, than the edge is removed, hence clusters are split [7].

The quality of the obtained clusters is often measured using the modularity. Modularity [14] is used to determine the quality of assigning a node to a community. It has a value between -1 and 1 and represents the density of the edges inside the community as compared to the density between communities. In [14] modularity is defined as:

$$Q = \sum_{c1}(E_{c1c2} - W_{c1}^2), \tag{2.2}$$

where E_{c1c2} represents the fraction of the total edge weight in the network that connects community c1 to community c2 and W_{c1} the total weight entering community c1.

Often, the links between nodes are directed. However, this is frequently omitted and graphs are considered undirected when applying community detection algorithms. This leads to losing valuable information about the network but eases the clustering technique. Another hurdle is detecting overlapping communities. It is often the case, especially in social networks, that an individual belongs to different groups. Traditional algorithms assign a node to only one community as detecting overlapping communities is not only not a trivial task, but also computationally heavy [7].

A good clustering algorithm manages to cluster vertices such that there is a large number of connections between the vertices in the same sub-graph and as few as possible between the sub-graphs. The procedure becomes more challenging when the network is dynamic [7]. A dynamic network is a network represented by a graph whose characteristics change frequently, i.e. nodes are added/removed or connections between nodes change. This is also the case for Universe, where connections can easily be changed, for example, when employees switch projects.

The problem of finding communities inside a graph is challenging and has not yet been completely solved as to match and satisfy every use case. With the increase of data this becomes even more difficult to accomplish.

2.1.2 Large scale clustering

In [23], Communities from Edge Structure and Node Attributes (CESNA), a community detection algorithm is presented. The algorithm takes into consideration the node properties and the network structures. Its aim is to detect overlapping, non-overlapping and hierarchically nested communities. The model is based on four properties: nodes that belong to the same

communities are likely to be connected, one node can belong to multiple communities, if two nodes have multiple communities in common they are more likely to be connected than if they share only one community and nodes belonging to the same community share common node attributes. The algorithm starts with a given set of vertices with attributes and a given set of communities. The community affiliations generate the edge structure. Moreover, they are also used to predict the node attributes. The work presented in [22] follows a similar approach but does not take into consideration the node properties. Both algorithms need an initial set of nodes labelled with a community affiliation. These two algorithms proved to be scalable and have been tested with production and social network data. Therefore, in order to use them in Universe, we should take the input data and assign each node to a default community. For example, people working in the same department would be assigned to the same community.

An interesting approach is presented in [5]. The authors introduce a game-theoretic framework to solve the community detection problem based on the social networks' structure. The strategic game is called *community formation game*. Given a social network, they consider each node as being selfish, that is, it selects communities to join or leave based on its own utility measurement. When each node has found its community, an equilibrium state of the game is reached. Each node can select more communities to be part of, therefore tackling the overlapping communities problem. The utility measurement of each node is computed using two functions, the gain function and the loss function. The gain function is based on Newman's modularity function [14] described before. The loss function is computed using the costs incurred when nodes join communities [5]. Therefore, a node has a benefit when joining a community, the benefit being computed by the gain function, but has to pay a membership to the community, amount computed by the loss function [5]. Each node is modelled as a rational and autonomous agent whose only purpose is to increase its utility function by joining the right communities [5]. In order to reach its equilibrium and output communities, the game tries to find local equilibrium. This means that no agent can deviate from its current strategy within the locally allowed space [5]. The actions an agent can take according to [5] are:

1. *Join* a new community.
2. *Leave* a community.
3. *Switch* from one community to another.

The algorithm was designed for undirected and unweighted graphs but can

be extended.

In [8], the authors compare various algorithms based on the LFR benchmark, a generalization of the GN benchmark introduced by Girvan and Newman in [10]. Their results show that the algorithms proposed by Rosvall and Bergstrom [20], the one proposed by Blonder et. al [3] and Ronhovde and Nussinov [19] have the best performance on the LFR benchmark and additionally have a low computational complexity, which makes them suitable for large networks. On the other hand, none of these algorithms can detect overlapping communities.

The algorithm presented in [20] works for weighted and directed networks. It is based on the probability flow of random walks. Maps are used in order to describe the dynamics of the links and nodes. A group of nodes which forms a structure where the information flows quickly, is considered to be a module [20]. Nodes are named using Huffman codes. This approach allows to assign short codewords to common objects and long codewords to rare ones. The length of the codeword for each node depends on the node visit frequencies of an infinitely long random walk. Using Huffman codes it is possible to describe the flow transitions of 71-step walk in 314 bits . With a uniform code, where the node name length is the same for every node, for a 71 step walk 355 bits are needed to describe the walk [20]. To create the map, the network is divided into two levels of description [20]. The unique names are used to name the modules, i.e. the course grained structures of the network, while the names associated with fine grained details are used to name individual nodes inside a module. This approach is widely known and it is used, for example, in assigning name streets on maps [20]. It is often the case where more cities have the same street names and this does not create any confusion. This approach has been tested on a network created by following citations among 6,128 journals in science and social science. The links between nodes represent citation flow, and their color and width depends on the flow volume. The resulted map can be observed in Figure 2.2. The size of each modules reflects the amount of time a random surfer would spend on following citations in that module [20].

In [19], a multiresolution algorithm is presented. The term "multiresolution" indicates that the algorithm is not limited to hierarchical structures. The aim of the algorithm is to determine and evaluate the relative strength of multiresolution structures. To do this, it examines the correlations among several independent replicas of the same graph over a range of resolutions [19]. A Potts model measure for community detection is used and applied to detect multiresolution structures [19].

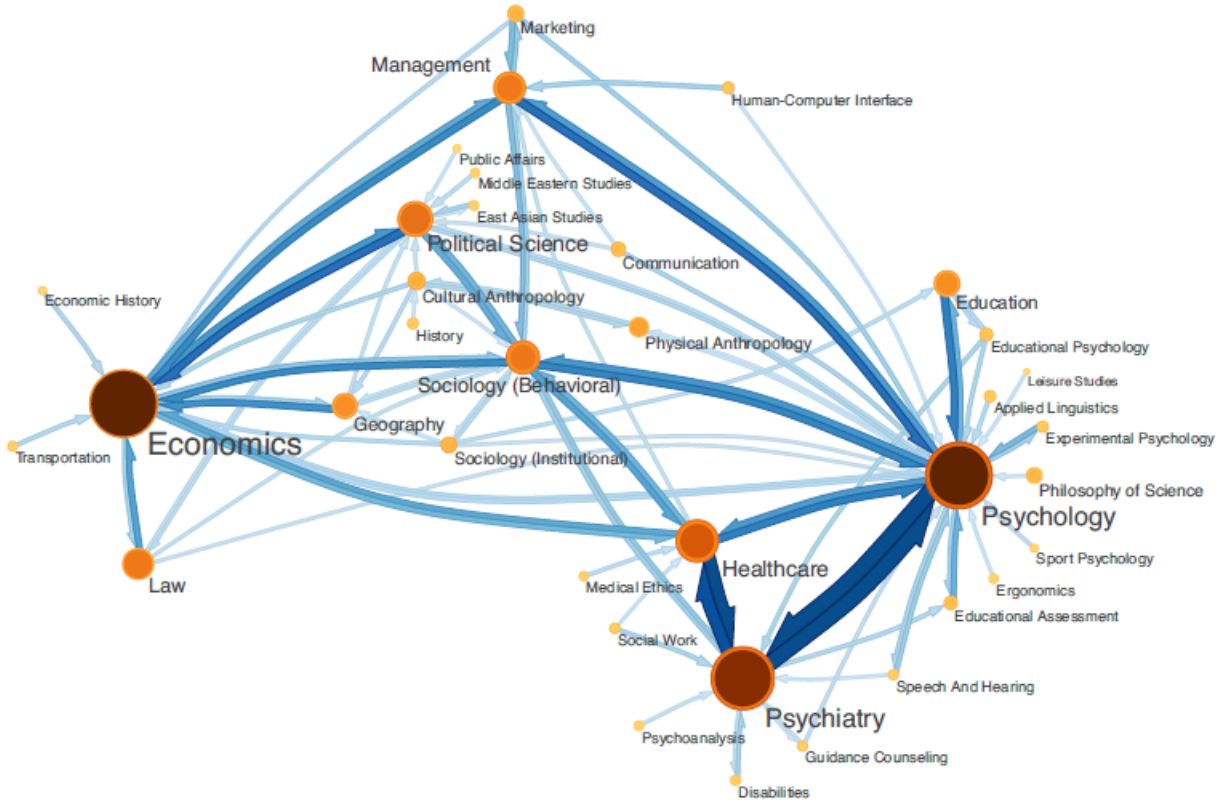

Figure 2.2: A map of the social sciences (taken from [20])

2.1.3 Detecting overlapping communities

A more realistic view upon community structure leads us to realising that the majority of them are overlapping nested structures. A person may belong to different social groups at the same time, depending on hobbies, work place, skills, etc. This does not apply only for social structures. For example, in biology, a large number of the proteins belong to several protein complexes simultaneously [9]. Clique percolation method (CPM) is a well known approach for analysing the overlapping community structure of networks. The communities are built up from k-cliques, which are fully-connected subgraphs of k nodes. A community consists of the union of k-cliques that can be reached from each other through adjacent k-cliques. Two k-cliques are adjacent if they share k-1 nodes. An example of an overlapping network can be seen in Figure 2.3.

In [9], a node i of a network is characterized by a membership number m_i,

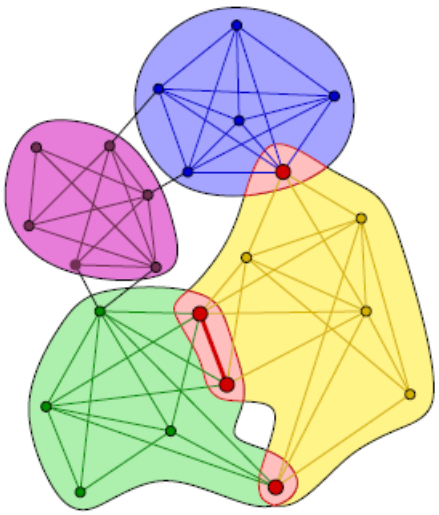

Figure 2.3: Overlapping k-clique-communities with k = 4 (taken from [9])

representing the number of communities node i is part of. The overlap size of two communities is the number of nodes the two communities share. The method presented in [9] is used for undirected and unweighted networks. The method consists of two steps. First, it localizes all maximal cliques of the network. Then, it identifies the communities by carrying out a standard component analysis of the clique-clique overlap matrix. A maximal clique is a clique that is not a subset of other cliques, i.e. it cannot be extended.

The work presented in [12] extends this procedure for weighted graphs and uses k as an input to the algorithm. Instead of finding all maximal cliques, the SCP algorithm tries to find all cliques of size k. The algorithm aims at detecting k communities by sequentially removing and then reinserting the constituent links to the network and observing the emerging structures [12]. For an unweighted network the order of reinserting the links is not important. However, for a weighted network, the links are reinserted in increasing order of their weight. The algorithm has two phases. In the first phase, the k cliques that form when a link is inserted are detected. Suppose a link is inserted between v_i and v_j. In order for a k clique to form, nodes v_i and v_j must have a degree of minimum k-1 [12]. The algorithm selects all common neighbour nodes of v_i and v_j. Each k-2-clique contained in this neighbourhood will give rise to a new k-clique [12]. All the newly detected k-cliques enter the second phase of the algorithm [12]. During the second phase overlapping k-cliques are found in order to detect k communities. A k community is

defined as a set of nodes that can be reached by a series of overlapping k-cliques [12]. The computational time of the SCP algorithm is linear and is dependent of the number of k-cliques in the networks [12]. SCP algorithm is suitable for low values of k. In [17] if two k-cliques share k-1 nodes, they are said to percolate each other and this is used in k-clique percolation method. The clique graph is a tool that can be used to understand the k-clique percolation technique. It is built by joining two nodes when their corresponding cliques overlap. However, it can be computationally expensive to build the clique graph [17]. In Figure 2.4 a clique graph can be observed. The initial network is represented by the blue circular nodes. The maximal cliques are represented by red squares. The connected component in the graph corresponds to the cliques that percolate each other [17].

The algorithms presented in [17] start by obtaining the maximal cliques in the graph. The first algorithm tries to build a minimal spanning forest over the obtained maximal cliques to reduce the unnecessary cliques overlappings. The second algorithm makes use of an additional hierarchical structure, which further reduces the number of full intersections. The goal is to find which cliques are in which connected components [17].

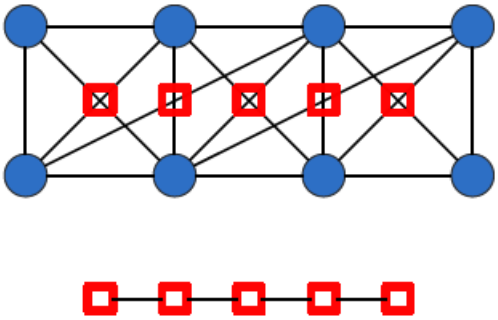

Figure 2.4: Initial network and its clique graph (taken from [17])

2.2 Overview of recommendation techniques

Recommender engines are built with the purpose of suggesting individualized, interesting and useful items to the users. This is what separates a recommender engine from an information retrieval one. Nowadays, almost any e-commerce platform tries to offer recommendations to its users. There

are three main components of a recommender engine according to the work presented in [4]. First, there is the background data. This represents the information that the system has before making any recommendations. In the Universe case, it can represent user's channels, user's working department, user's files or meetings. Second, there is the input data. This is the information the user must communicate to the system in order for it to make a recommendation. Finally, there is the algorithm. This combines the background data with the input data to generate a suggestion. Five different recommender techniques emerge based on these three components.

2.2.1 Main recommender techniques

The most widely used and well known technique is called **collaborative filtering** [4]. Inter-user relations are taken into consideration when making recommendations [4]. The technique identifies similar users based on their ratings and generates new recommendations based on the user comparison. In a collaborative system, a user profile is represented as a vector of items and their ratings [4]. The ratings can be represented as binaries (like/dislike) or a real values (e.g., levels of preferences). Users can be directly compared to each other, using for example, correlation. Models can be derived from previous ratings and used for predictions [4]. An advantage of collaborative techniques is that they are independent of the object being recommended (movies, books etc.) and work well regardless of the object being recommended [4]. A subclass of the collaborative filtering technique are the demographic recommender systems [4]. These systems classify users by their demographic position and personal information. The personal information is taken using an interactive dialog (interview) [4]. The responses to the interview are matched against a library of user stereotypes. The advantage is that a demographic system may not require to keep a history of previous ratings [4]. The collaborative filtering techniques are said to form people-to-people correlations [4]. A disadvantage comes with the reluctance of the user to share qualitative personal information [4].

A **content-based recommender system** learns the user's interests based on the features of the objects the user has rated, hence making item-to-item correlations [4]. Rather than computing similarities between the items the user liked in the past, the system computes similarities between users' interests. These interests are learned over time and adapted according to changes in users' preferences. Decision trees, neural networks, or vector-based representations can be used as learning methods [4].

Utility-based recommender systems make recommendations by computing the utility of a particular object for a specific user using an utility function [4]. In most cases, the utility function is represented by the user profile. An advantage of the methods is that the utility function can include non-product specific attributes, such as vendor reliability or product availability [4]. Hence, an utility based system, lets the user specify all the characteristics that matter for a recommendation made to him [4].

Knowledge based recommender systems attempt at storing knowledge about how a particular object satisfies a user's need. They gather this information through making the user answer a detailed questionnaire or building some preference function and reason about it. For example, Entree [4] uses knowledge about cuisines to reason about similarity between restaurants.

The **new arrival** is a well-known complication in recommendation systems. This problem is also called *"ramp up"* problem [4]. A newly arrived user, who has few ratings, is difficult to be categorized. Similarly, a new item, which has not been rated yet, is hard to be recommended [4]. In a collaborative filtering technique, where it is important for users to have overlapping ratings, another problem that emerges is when new items arrive very fast and the user base in not large enough to provide enough ratings to overlap. In conclusion, collaborative filtering is suitable for networks where there are many users with similar preferences. The utility and knowledge based recommender systems do not suffer from the *"ramp up"* problem, since they do not use previous statistical gathered data to make recommendations. However, their disadvantage comes when the user has to input preferences for a large set of data [4].

2.2.2 Hybrid recommender systems

A hybrid recommender system is built by combining two or more techniques presented before. In the majority of the cases, the collaborative technique is combined with another technique to overcome the "ramp up" problem [4]. According to the work presented in [4], there are different ways of combining techniques. The weighted approach starts by giving weights to the recommendations retrieved from each of the recommender engines. Then, it gradually adjusts the weights as the user gives feedback upon the received predictions.

The switching approach uses a criteria to switch between the recommender

techniques. For example, if the first technique is unable to make a prediction with a sufficiently high confidence, the system switches to using the second technique. This introduces an extra layer of complexity to the system as one has to determine the switching criteria.

In a mixing approach, recommendations from more recommender techniques are presented to the user. In the end, the user has the ability to select the most appropriate recommendation. In a feature combination approach, for merging collaborative filtering and content based, the collaborative information is treated as additional features in a content based approach. This makes the system aware of the collaborative data without being affected too much by the number of users who rated in item.

In a cascaded approach one recommendation technique is employed first, to produce a coarse ranking of recommendations and a second technique refines the recommendations [4]. The second technique only focuses on those recommendations that need additional discrimination. They do not affect the ones that have a high confidence level or a very low one from the first step [4]. However, it is important to keep in mind that the second recommender technique does not use the output of the first technique to produce its recommendations. The results of the two recommenders are combined in a prioritized manner [4].

On the other hand, in a feature augmentation approach, the second recommender uses some of the output of the first recommender to produce its result. In a meta-level approach the whole output of the first recommender is used as the input for the second one. Figure 2.5 presents the possible combinations of techniques in hybrid systems.

As one can observe from Figure 2.5, at the time of the work presented in [4], there were still plenty of unexplored combinations. Entree and its hybrid version EntreeC [4] are two recommender systems for finding restaurants. Entree uses a knowledge based technique, while EntreeC uses a cascaded approach of a knowledge based approach with a collaborative filtering approach. After making recommendations, in order to give ratings to the restaurants the system uses the following rating model:

1. *Entry point:* The user typed this restaurant in as a starting point so assume it is one they liked. Rating = 1.0 [4]..
2. *Exit point:* The user stopped so assume he did not find what he was looking for. Rating = 0.8 [4].
3. *Browse:* The user is browsing through the restaurant offers, does not

	Weighted	Mixed	Switching	Feature Combination	Cascade	Feature Aug.	Meta-level
CF/CN	P-Tango	PTV, ProfBuilder	DailyLearner	(Basu, Hirsh & Cohen 1998)	Fab	Libra	
CF/DM	(Pazzani 1999)						
CF/KB	(Towle & Quinn 2000)		(Tran & Cohen, 2000)				
CN/CF							Fab, (Condliff, et al. 1999), LaboUr
CN/DM	(Pazzani 1999)			(Condliff, et al. 1999)			
CN/KB							
DM/CF							
DM/CN							
DM/KB							
KB/CF					EntreeC	GroupLens (1999)	
KB/CN							
KB/DM							

(CF = collaborative, CN = content-based, DM = demographic, KB = knowledge-based / utility-based)

Redundant
Not possible

Figure 2.5: Possible and Existing Recommendation Hybrids (taken from [4])

input any critique but he does not order. Rating = -0.5 [4].

4. *Critiquing:* The user is giving negative feedback about the restaurant. Rating = -1.0 [4].

2.2.3 Profile representation

A recommender system cannot function without a user profile [13]. The system should gather relevant feedback about the user's behaviour, taste, or preferences. The raw feedback given by the user in a form has no sense by itself. The recommender system has to extract the information depending on the user profile [13]. Most commonly, the user preferences change with time. Therefore, a recommender system needs to have a technique to adapt

the user profile to new preferences and forget the old ones. There are several approaches when building a user profile, for example: history of purchases, web navigation or e-mails, an indexed vector of features, a n-gram, a semantic network, an associative network, a classifier including neural networks, decision trees, inducted rules or Bayesian networks, a matrix of ratings or a set of demographic features [13].

In the e-commerce field a common approach is to keep a history or purchases and user ratings as a user profile. Two popular recommender systems, Amazon.com and CDNow.com use this in order to characterize the user profile [13].

Another approach is to use a vector space model. Items are represented with a vector of features, each having an associated value. This value can be a Boolean, indicating presence or absence, or it can be a number, representing ratings, frequency or probability of the feature [13].

In weighted n-grams approach, items are represented with a network of words with weights in the nodes and edges, while semantic networks aim at storing the meanings and then utilize these meanings [13].

In collaborative filtering systems, it is common to keep a user-item ratings matrix. Each cell (u,i) contains the rating given by user u to item i. If a cell is empty, no rating has been given by user u to item i.

In demographic filtering systems the user profile is built based on stereotypes or "kind of user". The user profile contains a list of demographic features, which describe the user type [13].

2.2.4 Profile generation

One way of starting a user profile is by keeping it *empty*, that is no initial profile is generated. Another way is to use a manual system, which asks the users to input their preferences. A drawback of this approach is the unwillingness of the user to specify relevant information or the user not knowing which items are preferred. Stereotyping is another approach to initial profile generation. Usually, user profiles are generated based on their geographic location. Similar to the manual approach, the users might not trust the system and hence not provide their true location.

Another approach is to use a training set. The user is provided with a series of examples and asked to rate them as relevant or irrelevant to their preferences. A special attention has to be given to the chosen examples. In

order to obtain precise results, the set of examples has to be large enough and representative enough for the user.

2.2.5 Profile learning techniques

Using the initial profile data, a learning technique is generated. On the one hand, systems that acquire the user profile data from databases, systems that use collaborative filtering, systems that have an item-matrix as a profile representation or systems that use stereotyping to generate the user profile, do not need a learning technique [13].

On the other hand, systems where the information has no structure, need this learning technique. For example, when a book has to be recommended to a user, one approach is to use frequency word occurrence to calculate the relevance of the book. A well know technique is TF-IDF, that is terms that appear frequently in a document but rarely in other documents are the most representative for that particular document [13]. Classifiers is another method for structuring information and learning. They take as inputs the unstructured user profile data and data about an item and produce as output whether the item is interesting or not for the user. Different machine learning methods such as neural networks, decision trees, or Bayesian networks can be used to implement classifiers.

Finally, a hybrid approach is preferred to obtain user feedback [13]. In a hybrid approach, implicit feedback obtained from user rating or like/dislike items is combined with explicit feedback collected by the system, such as *"has the user clicked the recommended item?"*.

2.2.6 Profile adaptation techniques

As user's interests change over time, there is a need for an adaptation technique that will discard old information and take new one into account. There are various techniques presented in the literature, such as manually, a gradual forgetting function or the natural selection [13].

In a manual approach, the user changes the user profile when interests change. As in the manual initial profile generation approach, the main disadvantages are the effort asked from the user to keep the profile updated and the user not knowing exactly which his interests are.

Another approach is to extract data from the user feedback and adapt the profile to the answers [13]. In this approach old answers are forgotten, which in some cases might represent a loss for the recommender agent.

In a gradual forgetting function approach, the recommender system discards information older than a certain period of time and learning only from the observations taken in a particular time window [13].

The natural selection approach is implemented using genetic algorithms. Agents are responsible for making recommendations. Then, the agents with the best recommendations are selected and reproduced using crossover and mutation operators, while the rest of the agents are destroyed.

2.2.7 Profile item matching

In a content based approach, the items are compared directly to the user profile. Hence, techniques that compare the representation of the user profile with the representation of the items have been developed. Some of the techniques are keyword matching, cosine similarity, nearest neighbour or classifiers [13].

Keyword matching simply counts the common words between the specification of the user's interests and the specification of the items. A challenging problem here is to understand the context of the words and synonyms.

In a cosine similarity approach, both the user profile and the item are represented as n dimensional vectors [13]. The cosine of the angle between the two vectors is computed. If the result approaches 1, the item is recommended to the user, otherwise it is not.

In a nearest neighbour approach, the distance between the new item and the items the user has liked in the past is computed.

When using a classifier approach, the user profile can be represented as a neural network, Bayesian network, decision tree or any other classifier [13]. The items are categorized in classes: interesting, not interesting or not relevant.

2.2.8 Recommender Engine procedure

The basic steps of a Recommender Engine are: find similar users, create a neighbourhood, make a prediction based on the selected neighbours

[13].

Step 1: Finding similar users

Nearest neighbour is used not only in profile-item matching, but it can also be used to to compute similarity between users by analysing their preference history. Moreover, cosine similarity can be used in a similar approach as for matching profiles to items. Another approach is clustering. The user is assigned to a group of users based on some predefined stereotypes. Once the cluster is computed, recommendations are made based on the preferences of the cluster. Recommendations might not be personal enough in such a system but this approach is able to scale very well when there is a large set of users and items involved [13].

Step 2: Creating a neighbourhood

In order to create a neighbourhood some of the nearest neighbours have to be selected. Correlation-thresholding techniques, best n-neighbours or a centroid neighbours technique can be used. In a correlation-thresholding approach an absolute correlation threshold is set. Then, all neighbours with an absolute correlation greater than given thresholds are selected [13]. The centroid neighbours technique starts by selecting the closest neighbours to current user and to calculate the centroid between them. When new users are added to the group, the centroid is recalculated. More precisely, the algorithm allows for the neighbours to affect the formation of the neighbourhood, which is beneficial for sparse data sets. In a best-n-neighbours approach, n is a fixed value and for each user we select first n most similar users. Choosing a high value for n leads to having a lot of noise for users who have high correlates, while choosing a low value for n, leads to poor predictions for users with low correlates [13].

Step 3: Making the predictions

Some techniques to make recommendations based on the neighbourhood include: the most-frequent item recommendation, the association rule-based recommendation and the weighted average of ratings. In a most-frequent item recommendation approach, the engine looks at the user's interests and finds items that have been rated by the neighbours to satisfy these interests. At the end, the n most frequent items among all neighbours are selected and recommended to the user [13]. In an association rule-based approach, the engine infers rules previously generated from the neighbourhood rather than looking at each neighbour [13]. This approach is often combined with the most frequent item approach, as constructing strong rules is hard when the number of neighbours is low. The weighted average of ratings approach is

based on the assumption that all users in the neighbourhood give ratings of the same distribution. The engine uses correlation as a weight and computes the average of the ratings based on this weight [13].

2.2.9 Remarks

One can observe that there is a lot of work already done in this area. The problem of detecting communities inside social networks has already been addressed by many researchers interested in this area. In addition, techniques to make recommendations based on user clustering, more exactly, collaborative filtering, have been widely studied. However, there is still room for improvements. One of them is studying these methods in a real life environment, by implementing the proposed algorithms in a production application and obtain user feedback. Additionally, Universe needs a personalized Recommender Engine integrated in a workflow that can be used by any employee without prior knowledge of clustering or recommendation techniques.

Chapter 3

Environment

This chapter presents Universe, the Intranet platform used for employees' communication and collaboration, as well as other tools than can be used to process and visualize graphs.

3.1 Universe, the Google for Work Enterprise Intranet

Universe is an Intranet solution designed for companies using Google Apps. It integrates the whole Google Apps suite (Gmail, Google Drive, Google Hangouts, Google Calendar, etc), hence gathering all information regarding emails, drive files, calendar meetings and employees, in one place. The main components of Universe are: Channels, People and Pages.

The main page of Universe is illustrated in Figure 3.1. As it can be observed, a list of channels the user is part of is displayed on the left side of the page. **"My Stream"** is positioned in the center of the page and includes the latest posts from the channels the user is part of, displayed in a reverse chronological order. On the right side of the page, general news from the company are displayed. Last but not least, the **search** allows the user to look for Drive documents, employees, calendar meetings and emails.

A **channel** is a social workspace aimed to ease the communication inside teams. The user (usually an employee of the company) is able to create its own or participate in other colleagues channels. In the channel, the user can write posts or comment to other people posts. Moreover, all members of the

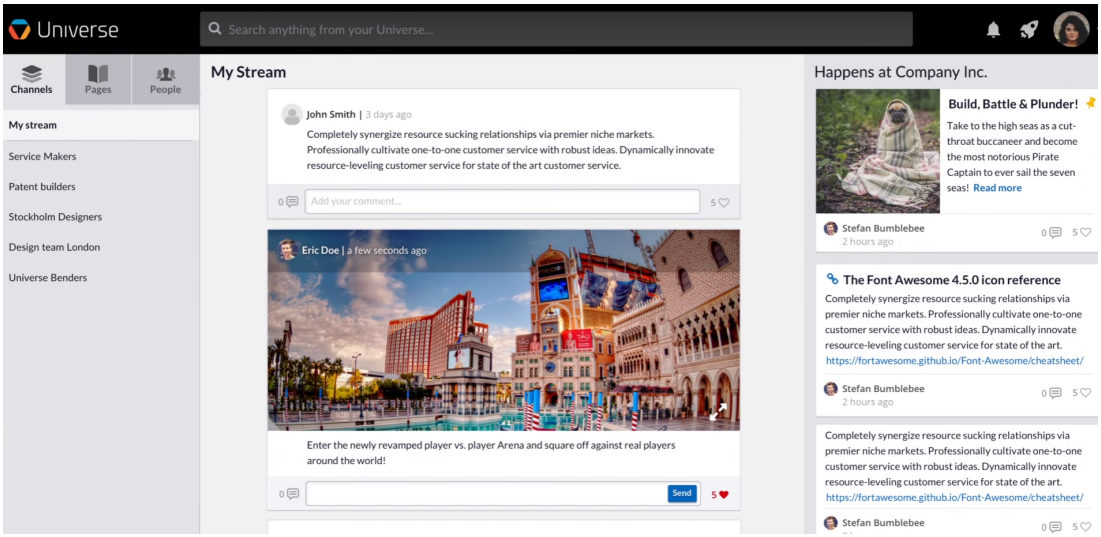

Figure 3.1: Universe Main Page

channel can share a Google Drive folder and a Google Calendar, making it easier to collaborate.

The **Pages** component incorporates all the statically accessed information, like HR materials or employees forms. Each user can define its own pages.

Using the **People** component, one can find and connect with people inside the company. Available information about employees including skills, account information or hierarchical position can also be retrieved.

3.1.1 Technology stack

Universe is deployed on Google App Engine and it is using Google Cloud SQL for storing data. Google App Engine (GAE) is a Platform as a Service (PaaS) cloud computing environment, which provides services for developing and hosting web applications in Google Datacentres. Google Cloud SQL is a service aimed to ease the management, maintenance and administration in the cloud of a MySQL relational database. Universe uses Java and Spring Framework for the backend development and AngularJS, HTML and CSS for the frontend. It also uses multiple Google APIs to integrate Google Apps tools (e.g., Calendar, Drive). The backend and the frontend are strongly decoupled. The backend exposes a REST API. The API is created using Google Cloud Endpoints. Google Cloud Endpoints consists of libraries and tools that allow to generate APIs from an App Engine application. It uses

annotations to describe the API configuration, methods, parameters, and other details that define the properties and behaviour of the Endpoint. The main annotation is @Api. This annotation makes all public and non-static methods of a class, exposed in the public API.

3.2 Community detection tools

This section introduces the community detection tools used in this thesis work for visualizing and exploring data.

3.2.1 Gephi - The Open Graph Viz Platform

Gephi[1] is a software that permits graph visualization and exploration. Gephi can be used to:

1. Manipulate networks in real time.
2. Detect structures of the network.
3. Detect patterns in data.

Gephi uses an OpenGL engine to allow real time graph visualization. It supports up to 100,000 nodes and 1,000,000 edges networks. Layout algorithms can be applied to shape the graphs. Gephi integrates some state-of-the-art layout algorithms. Gephi comes with a statistics and metrics panel, which allows performing different types of analysis on the graph, such as: Betweenness Centrality, Closeness, Diameter, Clustering Coefficient, PageRank, Community detection, Random generators and Shortest path. Nodes and edges can be filtered dynamically based on their properties. Colors, sizes, node and edge labels or fonts can be customized. Graphs can be exported as PDF, SVG or PNG. Graphs can be imported into Gephi using CSV, GML files or relational databases. There is a large variety of plugins that extend Gephi's functionalities. For example, the Neo4j plugin allows importing a NoSQL graph database into Gephi. An overview of Gephi user interface can be seen in Figure 3.2.

Gephi uses Louvain clustering algorithm [3] to detect communities. Another available plugin for community detection is Label Propagation Clustering (LPA). It supports weighted edges and provides implementations for three

[1] `https://gephi.org/features`

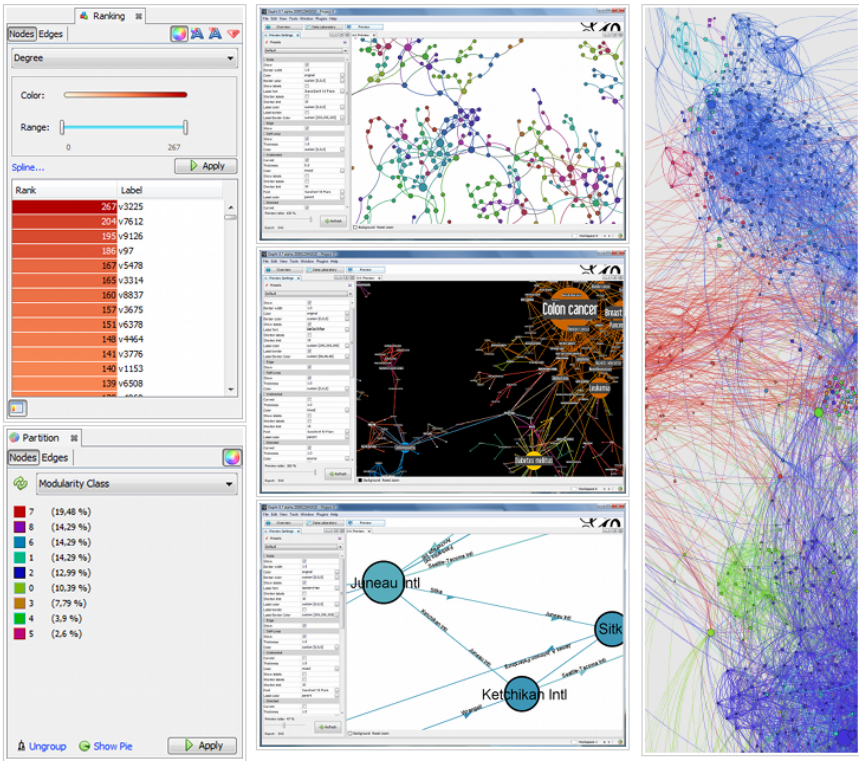

Figure 3.2: Gephi

LPA variants: basic LPA [16], randomized LPA (LPAr) [1] and modularity maximization LPA (LPAm) [1], each described by a specific propagation rule.

3.2.2 CFinder - Overlapping dense groups in networks

CFinder[2] is a tool for finding overlapping communities based on the Clique Percolation Method [9]. After finding all maximal cliques it generates the overlap matrix between these cliques. Once the matrix is computed, it calculates percolations for all values of k (where k is the number of nodes in a fully connected subgraph). Building the full overlap matrix equivalent to the full clique graph requires $O(n^2c\ )$ clique-clique comparisons, where nc is the number of maximal cliques [9]. The communities can also be visualized using CFinder's user interface Figure 3.3.

[2] `http://www.cfinder.org/`

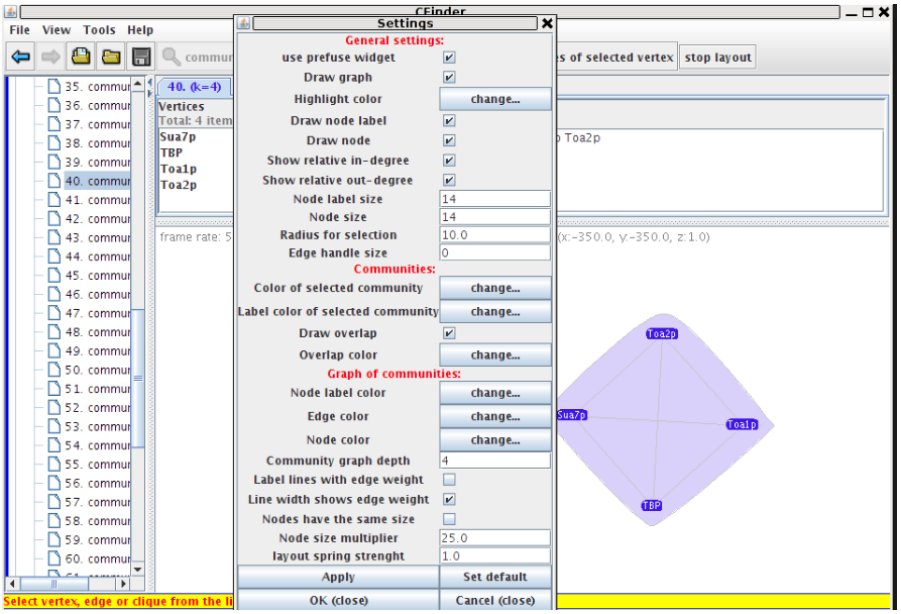

Figure 3.3: CFinder

3.2.3 NoSQL Databases for storing graphs

Graph databases are an alternative to relational databases for applications where data needs to be represented in a much more natural form. In a graph database the relationships between entities is the driving force of the data model. As this is the case for Universe Community Recommender, where we are particularly interested in the relationship between users, a graph database is the most suitable. The property graph is an attributed, labelled, directed multi-graph and it is supported by most graph databases [18]. The property graph is the most complex type and all the other types of graphs are subtypes of the property graph. Therefore, a property graph can model all the other types of graphs (e.g., undirected, single-graphs). Finding one nodes' relationships becomes faster when using a graph database. In a traditional SQL approach, each traversal of an edge in a graph would be a join, which can be an expensive operation especially if the data structure is recursive and complex [18].

3.2.3.1 Neo4j

One example of a graph database is Neo4j, a NoSQL, ACID compliant graph database. ACID database means it respects the properties that guarantee

the reliable execution of the transactions: Atomicity, Consistency, Isolation, Durability. Neo4j uses its own query language, **Cypher**, to execute queries. Cypher resembles the SQL query language but it is centered around matching patterns in a graph. Cypher queries begin with a **MATCH** clause that tries to match a pattern against the graph. It is followed by a **WHERE** clause, which provides additional filtering of the result. Neo4j can be deployed as a standalone server or as an embedded database, which gives developers access to its Java API. An easier and more similar to relational database development is the Spring Data for Neo4j library [3]. Spring Data Neo4j offers functionalities to annotate the entity classes with specific annotations(e.g., @Vertex, @Relationship) and map them to the Neo4j database model. Neo4j can be configured in High Availability (Neo4j HA) mode. The typical architecture will consist of a master node and a number of slaves nodes. A write to one slave is not immediately synchronized to the all other nodes, loosing consistency for a period of time. That is why HA mode is mostly used for increasing the capacity of reads. In the case of Universe Community Recommender, the writes (updates of communities) will only occur as a batch job or on non-frequent demand, while the reads will be frequent. Therefore, Neo4j in a HA mode is a suitable design for the Universe Community Recommender. Neo4j uses Apache Zookeeper [4] to coordinate nodes.

3.2.3.2 Google Cloud Datastore

Google Cloud Datastore [5] is a NoSQL document database that offers ACID transactions and was designed for automatic scaling and high availability. Data objects (i.e. entities) have one or more named properties. Each property has one or more values. Entities of the same kind can have different properties, and entities of different kinds can have properties with the same name but different value types. Each entity is uniquely identified by a key. An individual entity can be queried after this key or more entities can be retrieved by constructing a query based on the entities' keys or property values. At the application level, Google Cloud Datastore provides the Datastore API, which can be used to create, retrieve, update, and delete entities. Cloud Datastore is not suitable for storing analytical data or large blobs of data.

[3] `http://docs.spring.io/spring-data/neo4j/docs/current/reference/html/`

[4] `https://zookeeper.apache.org/`

[5] `https://cloud.google.com/datastore/`

Chapter 4

Methods and system description

This chapter describes Universe Community Recommender. The system detects communities inside Universe and makes recommendations to users based on their communities and interactions with the platform. Universe Community Recommender is composed of two main modules: Community Detector and Recommender Engine.

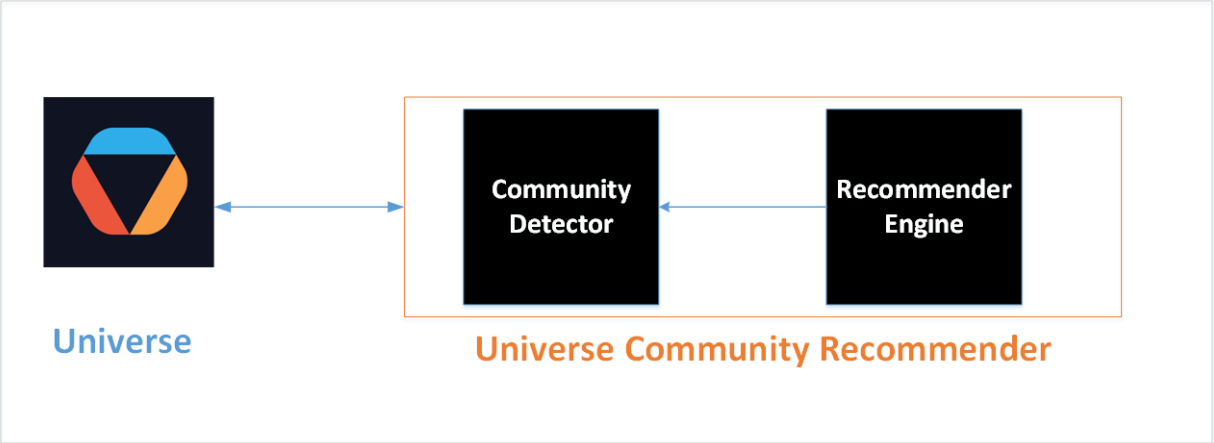

Figure 4.1: System components

4.1 Input data

The system collects as less data as possible, as this is a sensitive topic in companies. Moreover, most of the collected data is not stored and once

the weights between users are computed, it is discarded from memory. The system builds an adjacency matrix from the Universe data, then feeds this matrix to the community detection algorithms. Finally, based on the communities found, it outputs recommendations for each user.

Users are represented as Vertex objects. The collaboration between two users is computed by quantifying interactions between them. More specifically, the system uses user data, Channel data, Google Calendar Data and Google Drive Data and sums up the weights computed from each, as expressed in formula 5.3.

The collected user data includes user's name, email or other identifiers. This is needed for the identification of each user. In Universe, the user is identified by name, email and id. These can easily be adjusted and other types of identifiers can be defined. This identification data is used to create a Vertex object as described in the code snippet 5.1 for each user. The graph will be saved in the Neo4j database. The system has been visualized with Gephi Graphic UI and Neo4j Web UI.

To compute the edge weight between vertices, channel data, Google Calendar data, and Google Drive data is used.

In case of channel data, the system looks at each pair of users and finds the number of common channels between the two. The channel weight is directly proportional with the number of common channels. Data from liking and commenting on each other posts is also added. We consider that users who frequently comment on each other's posts and like each other's posts are more connected than the rest of the users. Therefore, weights are added for this type of interactions as well.

As for Google Calendar data, the system collects data about each events' participants. For each pair of users, it finds the number of common events they are part of. The Google Calendar weight is directly proportional with the number of common meetings the users have.

Finally, Google Drive weight is added to the sum of weights. More specifically, the system looks at the common modifications made by pairs of users on Google Drive files. For example, if vertex *v1* and vertex *v2* have both edited a Drive file, a weight to the edge weight between the two is added. Again, the Drive weight is directly proportional with the number of common modifications on Drive files.

The software will not retrieve or store Google Drive file content or Calendar meeting descriptions. The software will only store identification data for each

vertex and the computed weight between the users. The **My Workbench** section (part of the Recommender Engine) will display calendar meeting dates and participants, drive file titles, modification dates, and access links to them. This data is queried from Google APIs and it is not stored in the Universe database.

4.2 System architecture

Universe Community Recommender (UCR) is an independent module of Universe. UCR needs access to users' details (department, email, etc) and user interactions to compute the weights. In Figure 4.2 we present the interactions between Universe and Universe Community Recommender and the main components of the system.

The UCR most important components include: the VertexAdapterService, the two clustering algorithms that transforms user's data into vertices and the Neo4j service, which calls the REST Endpoints exposed by the Graph Data Manager to save or retrieve communities. The Graph Data Manager exposes a REST API to write and retrieve data from the Neo4j database. It also performs some business logic checks before saving data to the database.

4.3 Use cases

The HR administrator is able to start the community detection algorithm or schedule it at a specific point in the future. The request parameter for this call is the community detection approach and the user can choose between:

1. **Louvain Algorithm:** This option will apply Louvain algorithm on the graph constructed from the company's data.
2. **MCL Algorithm:** This option will apply MCL algorithm on the graph constructed from the company's data.
3. **Gephi:** This option will generate a .gml file that can be imported into Gephi (see Section 3.2.1) for further graph exploration or for applying Gephi community detection algorithms on it.
4. **Universe:** This will output the top three friends for each user based only on the adjacency matrix.

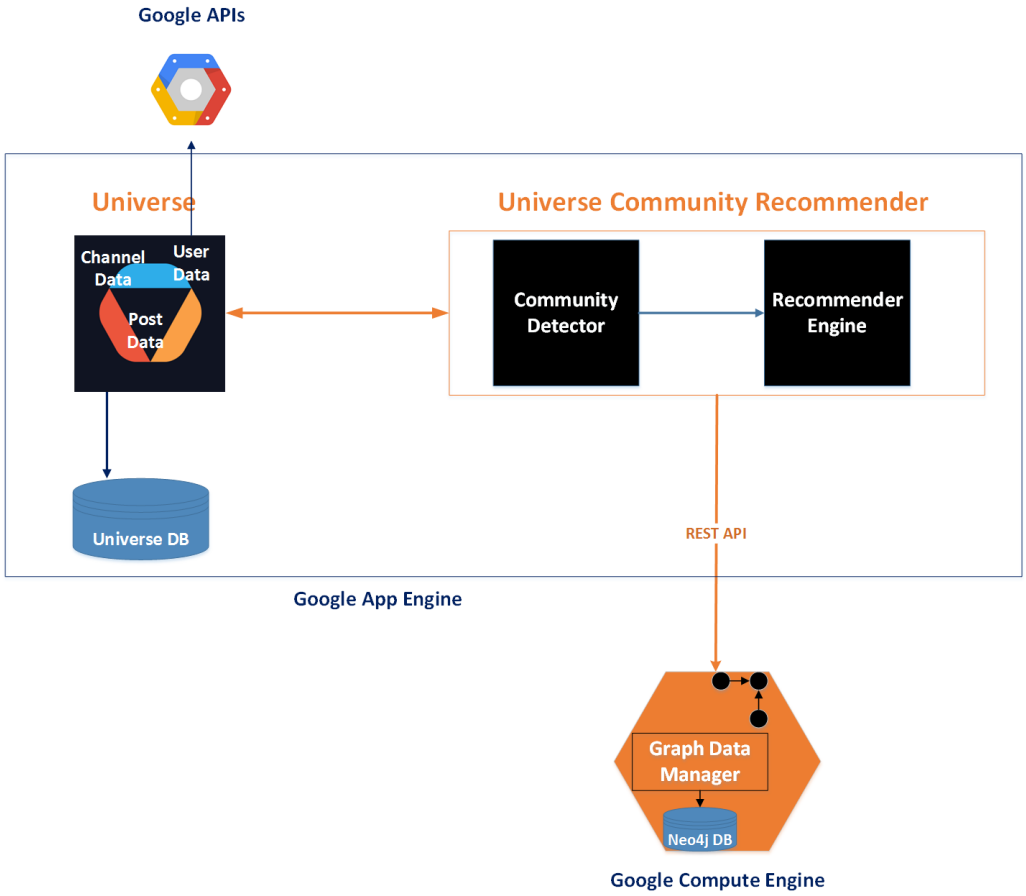

Figure 4.2: System Interactions

The system automatically detects the company the HR administrator belongs to, and only retrieves users that are employees of that company from the Universe database.

Moreover, the HR administrator has the possibility to schedule the community detection module at a certain date and time. One possibility is to recalculate the communities every night. In this way, the system is up to date with the interactions from the previous day. Having a real-time recalculation of the communities for each user request would put an overhead on the whole system and affect the response time for all the other requests. That is why, the communities are computed in a separate request, preferably during the night, when the system is less loaded.

The Universe social graph could be used by the company's HR administrator to analyse the structure of communication in the company, to detect isolated

users, to identify key users and to form better teams. Figure 4.3 presents the HR administrator's use cases.

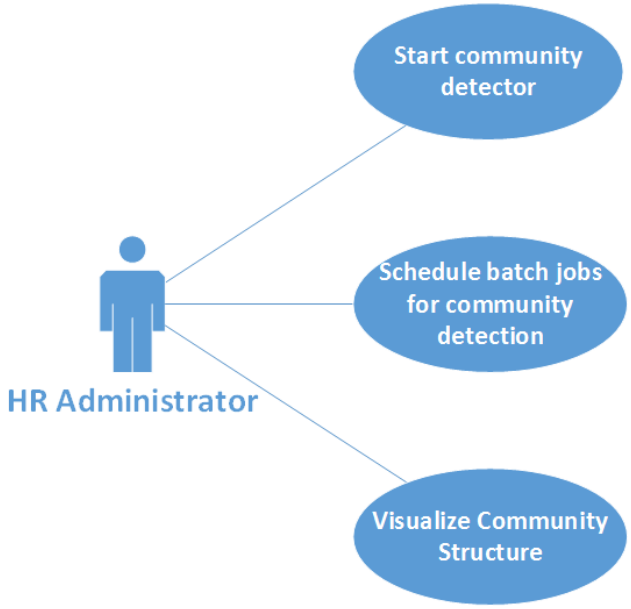

Figure 4.3: HR Administrator Use Case Diagram

Some use cases derived from the community detection include adjusting user's feed according to the user's community posts or automated suggestions of channels with users belonging to the same community. In addition to the aforementioned, the system generates a "**My Workbench**" panel, where information regarding scheduled appointments for the current day, last edited Drive files, unread emails and three most connected users is displayed for each user. This enhances the user's experience with the software and eases the access to information.

Displaying information regarding scheduled appointments for the current day, getting last edited Drive files or unread emails is computed independently of the communities and is concerned only with the user's own data.

In case of personalizing the user's feed, if user x is strongly related to user y, then user x feed will display first the most recent posts of user y. For the channel suggestion recommendations, the software sorts the open channels in increasing order of the number of participants that belong to the current user's community. For example, if channel a contains 10 participants from user x community, while channel b contains 20 participants from user x community, then the displaying order of the channels will be b, a. Figure 4.4 presents the regular user's use cases.

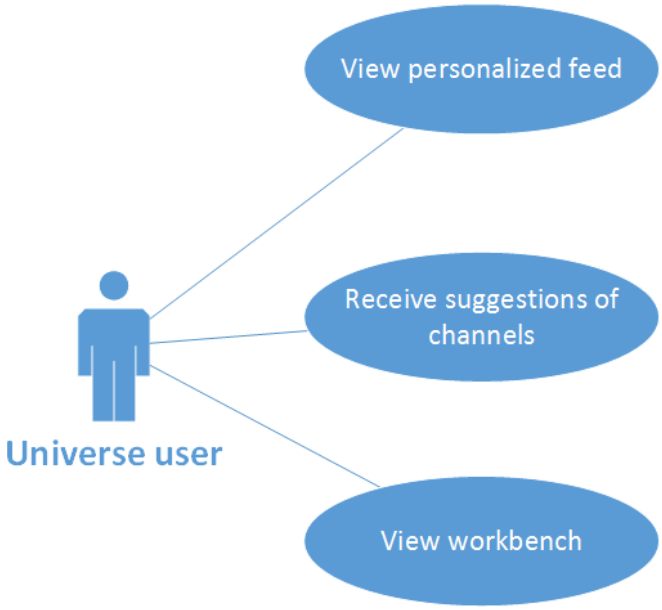

Figure 4.4: Universe User Use Case Diagram

Last but not least, search could be improved by first displaying results that might interest the user. This is considered as an extension of the current implementation of the Recommender Engine.

4.4 Application workflow

We separate the HR administrator's workflow and the regular user's workflow. On the one hand, the HR administrator is responsible for maintaining the communities up to date by either running the community detection module frequently enough or by automatizing the process and scheduling it as a batch job. On the other hand, the regular user does not have to perform any kind of task to benefit from the Recommender Engine. The user's "My Stream" section will automatically adjust to the user's community, while channel creation suggestions will appear on the web interface. Last but not least, "My Workbench" is accessible for every user to reduce the time spent to find important information.

4.4.1 Step one: Community generation

The HR administrator (or batch job) makes a request to Universe. The Universe forwards this request to the Universe Community Recommender. The Community Detector computes the communities by applying the following operations:

1. Makes a GET request to Universe for user information, Channels, Calendar and Drive data.
2. Converts each user to a vertex by storing user's id, user's name and user's email.
3. Applies the algorithm to detect communities.
4. Makes a POST request with the community structure to the Graph Data Manager (deployed on a separate Google Compute Engine instance).
5. The Graph Data Manager writes the received data to the Neo4j Database.

After this process the communities are stored in the Neo4j database.

4.4.2 Step two: User recommendations

Each time a user logs in or refreshes the page, the My Stream will display the most recent posts out of which the ones posted by the user's community will be displayed first. The user has the possibility to go to "My workbench" section and see useful information extracted by the Recommender Engine from the user's actions. Last but not least, the user will receive a list of suggested Channels to join.

4.4.3 Technologies used for implementation

The Universe Community Recommender has been implemented using Spring Framework. It is deployed on Google App Engine alongside Universe. It has endpoints for:

1. Create communities (called by the HR administrator only).
2. Get my communities.
3. Get personalized feed.

4. Get my workbench.
5. Get suggested channels.

The Graph Data Manager is implemented using Spring Boot Framework and it is running on a Compute Engine, configured with a public IP. The Graph Data Manager is connected to a standalone Neo4j Server installed on the same machine. The Neo4j web interface is publicly accessible, hence the HR administrator can make use of its visualization tool. The Community Detector and the Recommender Engine are implemented using Java 7 and Spring Framework. The "My Workbench" section uses Google APIs to retrieve unread emails, Google Drive files and calendar events. Gephi UI, Neo4j Web UI and cFinder UI are used to visualize and further process the results.

4.5 Roadmap

In the first stage of the project, the Universe user will have a daily recommendation digest containing people with whom the user will work, meetings, documents that the user is likely to use that day and unread emails. Moreover, a privileged user will have access to the social graph of the company and will be able to interactively visualize and query it. The user will be able to label the information received in the digest in helpful or unhelpful. This will help our software to adjust and improve by learning user preferences.

Chapter 5

Community Detector

This chapter describes the approach used to cluster users inside Universe based on their digital and social actions. There are two sources of data that can be used when clustering a graph, node properties and graph structure (i.e. interactions between the nodes). In most clustering algorithms only one of these sources is used. For example, node properties in a social network could be profile information, while graph structure could be build by following nodes friendship relationships. Community Detector combines the two sources of information as we need to consider both user information, and interactions between the users.

5.1 Procedure

The communities are detected among users belonging to the same company domain, hence data from different companies is not mixed. Each company will be represented by an undirected weighted graph. The community detection module is composed of three pipelined phases as it can be observed in Figure 5.1.

At the end of the initial phase, vertices are not connected to each other (Figure 5.2). A squared weighted adjacency matrix is created for these vertices. The dimension of the matrix is equal to the number of vertices. The weighted adjacency matrix is initialized with 0 for all values (Figure 5.1).

The second phase is responsible for computing weights for each slot in the matrix. The weight of the edge between vertex *v1* and vertex *v2* depends on the collaboration between the users represented by vertex *v1* and *v2*. If

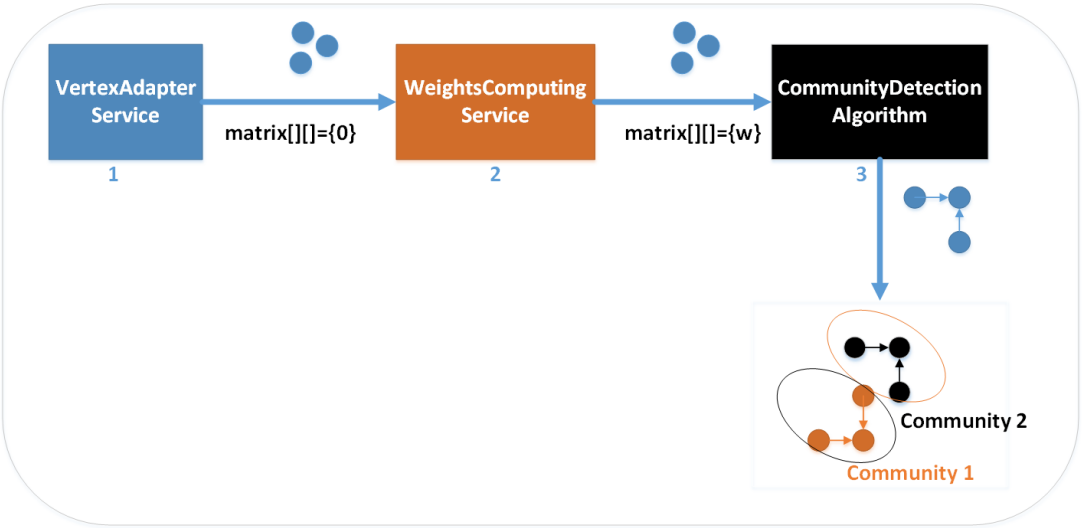

Figure 5.1: Community Detector

there is no collaboration between the two users, the weight remains 0. At the end of the second phase, each slot i, j in the matrix will be a number representing the weight between vertex_i and vertex_j.

Finally, the weighted adjacency matrix is supplied to a community detection algorithm (in our case Louvain or MCL). The output of this phase is a set of communities (Figure 5.1).

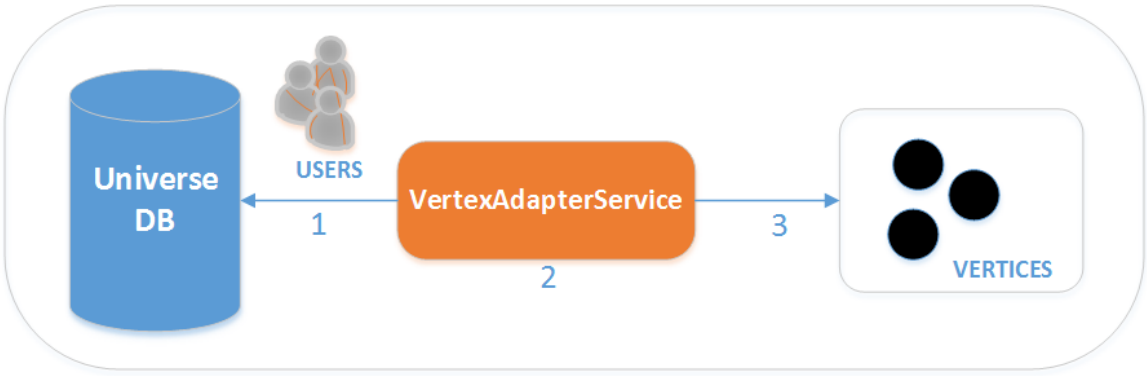

Figure 5.2: Phase 1: Converting users to vertices

5.2 Algorithms description

Graph clustering approaches aim to group vertices of a graph into clusters, based on their edge structure and edge weight. The ideal case is when there are many edges inside one cluster but as few as possible between the clusters. These clusters can also be called communities or subgraphs of the graph. Detecting communities in graph is a widely researched subject as it brings benefits to many areas where data can be represented as a graph such as biology, psychology or computer science. In order to detect communities inside the Universe platform two algorithms have been used.

5.2.1 Markov clustering algorithm

The Markov Clustering algorithm [21] is a graph clustering algorithm based on the simulation of a random flow inside the graph. A Markov chain is a stochastic process in which the next state is independent on the past states. The next state only depends on the current state. The probabilities to move to the next state from current state form the transition matrix of the Markov chain. Markov chains can be represented by a weighted directed graph. The vertices in the graph represent the states and the edges represent the transitions between states. The edge weight corresponds to the probability of moving from one state to the other.

We start by creating a transition matrix. The matrix will consist of probabilities to go from node i to node j as seen in Figure 5.3.

Being a probability matrix, the sum on each column is equal to one. If the graph is weighted, as in our case, the matrix contains the weights instead of probabilities. Once the matrix is built, the algorithm has six steps:

1. *Normalization:* Each value in the matrix will be divided by the sum of all values on its corresponding column.
2. *Add self loops:* For each node add a 1 on the main diagonal of the matrix so that the mass does not increase drastically when expanding with odd powers.
3. *Expansion:* The e^{th} power of the matrix is computed, where e is an input parameter called the expansion factor.
4. *Inflation:* Each element in the matrix is squared (if inflation operator is 2) and then the matrix is normalized again.

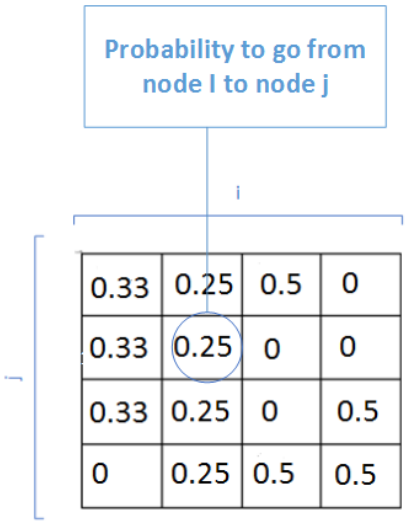

Figure 5.3: Probability Transition Matrix

5. Repeat Expansion and Inflation until the matrix converges (matrix's elements do not change anymore).

6. Interpret the results.

Figure 5.4 shows an example of a converged matrix. To interpret the detected clusters we look at each row and select the elements that are maximum on their column. In our example, the elements $1, 4$ will be one community and elements $2, 3$ will be the other community.

1	-	-	1
-	0.5	0.5	-
-	0.5	0.5	-
-	-	-	-

Figure 5.4: MCL Interpreting Clusters

5.2.2 Louvain clustering algorithm

The Louvain method for detecting communities [3] is an algorithm that relies on maximizing the modularity metric. After assigning the node to the community, modularity techniques compare the density of the edges inside the community with the density before assigning the node. Louvain algorithm is iterative and has 2 phases. The algorithm starts with assigning a community to each node in the network, hence after this initial step the number of nodes equals the number of communities. In the first step, the algorithm selects a random node and checks for which neighbour the modularity metric is maximum when this node is moved to the neighbour's community. If the modularity value is positive and maximum among all the node's neighbours, the node is assigned in its neighbour community. In a greedy manner, this step is repeated for each node, each time selecting the local optimal configuration (i.e. the one with the best modularity) until no further improvement can be obtained. The order in which the nodes are selected affects the computational time of the algorithm [3]. The gain in modularity by moving node i in community C is computed in [3] using the formula:

$$\Delta Q = \left[\frac{\sum_{in} - k_{i,in}}{2m} + \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right], \tag{5.1}$$

where $\sum_{\text{in}}$ is the sum of the weights inside community C,

$\sum_{\text{tot}}$ is the sum of the weights of the edges incident to nodes in community C,

k_i is the sum of the weights incident to node i, and

k_{in} is the sum of the weights of the edges from i to nodes is C and m is the total sum of the weights in the network

The next step converts the communities found in the previous step into nodes. The edges between nodes in the same community are converted in self edge loops. The edge weight between two communities is computed as the sum of the edge weights between the nodes in the two communities. The two steps are repeated until there is no modularity improvement. The algorithm is easy to understand and implement and has a linear complexity on typical and sparse data [3]. The number of communities decreases drastically after a few steps. One limitation of the algorithm is the memory storage, as the best performance is achieved when the entire network has to be loaded into memory. One of the limitations of Louvain method is that is does not detect overlapping communities. Another problem with modularity optimization

algorithms is the poor detection of small communities in large networks.

5.3 Implementation

In this section, the implementation of the main components of the Community Detector are described.

5.3.1 Universe Graph Model

The Universe Graph model is simplistic and intuitive and has been created to respect the Neo4j programming model (see Section 5.3.4). A class diagram is presented in Figure 5.5. The Vertex contains a list of edges. These edges represent the Vertex's community. This list is populated after one of the clustering algorithms is run. An Edge object is defined by a weight, a source and a destination vertex.

Figure 5.5: Universe Graph Model

To create the Vertex objects, the **VertexAdapterService** takes as input the users belonging to the organisation for which the community structure is computed, and creates Vertex objects containing only relevant information (name, email, id) from the users' data.

```
1 public List<Vertex> createVerticesFromUsers(
List<User> users){...}
```

5.3.2 Adjacency matrix

The adjacency matrix is a square matrix used to represent a graph. The non zero elements of the matrix indicate that the vertices corresponding to the

(i,j) position are adjacent. In this case, the adjacency matrix is a weighted matrix, which means the (i,j) position will store the weight between the vertex corresponding to index i and the vertex corresponding to index j. Each characteristic or action shared by two users is measured by a numeric weight. The weights have been set through an iterative process of weight adjustment. These can be configured depending on each company case by changing the **Weights** enum object. The actions and their weights for one test company are:

```
BOTH USERS ARE IN THE SAME DEPARTMENT = 0.7
BOTH USERS ARE IN THE SAME CHANNEL = 1
ONE USER COMMENTS ON THE OTHER USER POST = 0.5
ONE USERS LOVES THE OTHER USER POST = 0.3
ONE USER COMMENTS ON A DRIVE FILE EDITED BY
THE OTHER USER = 1.3
BOTH USERS EDIT THE SAME DRIVE FILE = 1.5
BOTH USERS COMMENT ON THE SAME DRIVE FILE = 1.2
BOTH USERS ARE PARTICIPANTS IN THE SAME MEETING = 2
```

We define *v1* as a vertex represented by user *x* and *v2* as a vertex represented by user y. They both belong to company z. The interactions that form the edge weight between vertex *v1* and *v2* defined so far are: *v1* and *v2* having shared channels, actions on the same posts (replies from *v2* to *v1*'s posts or vice-versa, love posts from *v2* to *v1*'s posts or vice versa), *v1* and *v2* editing or commenting on the same Google Drive files, *v1* and *v2* being participants in the same Google Calendar meetings. Each interaction has an initial assigned weight, which is adjusted through experiments and user feedback.

When computing the shared channel interaction, the software collects only the number of channels in which *v1* and *v2* are participants and computes a channel weight. For example, if *v1* and *v2* have n shared channels, the total channel weight will be computed as:

$$totalChannelWeight = n * sameChannelWeight, \tag{5.2}$$

where sameChannelWeight has an initial assigned weight of 1. No data regarding the channel content is stored or used. The same approach is used when computing the weight for the other interactions. The software counts the number of replies or love posts of *v1* to *v2*'s posts and vice versa, the number of Google Drive files edited by *v1* and *v2* and the number of Calendar meetings between *v1* and *v2*, and multiples these with their assigned

weight. Hence, the total edge weight between two vertices will be computed as:

$$edgeWeight = totalChannelWeight + totalPostWeight + totalDriveWeight + totalCalendarWeight \quad (5.3)$$

The total weight between two vertices is computed using the aforementioned formula 5.3. A sample of the matrix is presented in Figure 5.6.

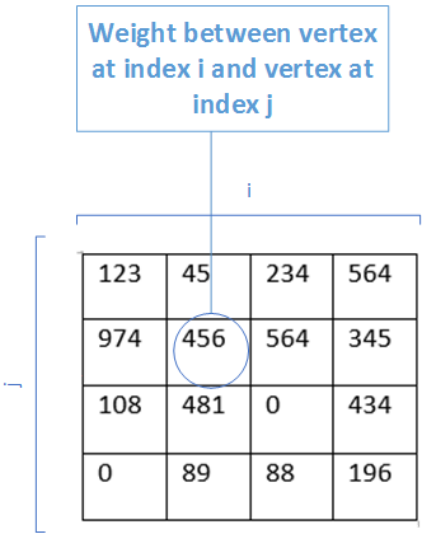

Figure 5.6: Sample of the weighted adjacency matrix

5.3.3 Community Detection Module

The Community Detection Module takes as input the adjacency matrix described before and outputs the communities detected by the clustering algorithm. Once the matrix is created, it can be passed to one of the algorithms Louvain or MCL. Computing communities based on the adjacency matrix is very convenient as the adjacency matrix is one of the most used models to represent a graph. Each algorithm has a corresponding implementation class (LouvainAlgorithm.java and MCLAlgorithm.java) that extend the ClusteringAlgorithm abstract class.

The application is easily extensible, hence new algorithms can be plugged in without changing the existing implementation. The new clustering algorithms must implement the abstract method:

```
abstract List<Cluster> cluster (double [][] adj_matrix, List<
    Vertex> vertices)
```

In case the new algorithms do not need an adjacency matrix, the vertices list can be used instead. This would be the case for an algorithm that only uses node properties to detect communities.

The result of the cluster method is a list of Cluster objects, which is a simple class composed of a list of connected Edges. Therefore, each community will consist of a list of connected edges. The Cron Service is used for scheduling the community detection as a background job. A background job is a task performed without a user request. They are not initiated by a user but configured by the application to perform certain periodic tasks (e.g., periodical reports, nightly backups).

5.3.4 Graph Data Manager

Graph Data Manager is an extra layer over the Neo4j Server. It exposes a REST API, which is called bu the Universe Recommender Engine. The main methods are:

```
public void saveVertices(@RequestBody List<Vertex>
vertices) {...}
public List<String> getCommunity(@RequestHeader
(name="email") String email){...}
public List<String> getTopFriends(@RequestHeader
(name="email") String email) {...}
```

The **saveVertices** method is called once the algorithm has finished computing communities. It takes a list of vertices and saves them to the Neo4j database. Note that each Vertex object in the list contains a list of edges representing their teammates. The **getCommunity** method searches for the vertex corresponding to the email address received as parameters and returns the vertex's connected vertices (i.e. its community). The **getTopFriends** selects the three edges with the largest weights among the vertex's community. The project has been implemented using Spring Data Neo4j. This tool enables POJO based development for the Neo4j Graph Database using Spring programming model. Annotations are used to map classes and attributes to

the graph structure. The **@NodeEntity** annotation specifies that a POJO class is an entity that represents a node in the graph database. To specify properties for the Vertices **@Property** annotation is used. Last but not least, specifying relationships is done though **@Relationship** annotation. In the following snippet, the annotated Vertex class is presented.

```
@NodeEntity (label="User")
public class Vertex  {
  private Long id;
  @Property
  private String name;
  @Index(unique=true)
  private String email;
  @Index(unique=true)
  private Long universeUserId;

  @Relationship(type="WORKS_WITH",
  direction = Relationship.OUTGOING)
  private  List<Edge> teammates;
```

Listing 5.1: Vertex representation

5.4 Cron Jobs

In order to schedule the community detection request, a Cron Job that runs every night is scheduled. *Cron* is a Linux utility that allows users to schedule a task (usually a script in Linux) to run automatically at a specified time and date. The Cron job is the scheduled task. Cron jobs are useful to automate repetitive tasks. As the community detection is a repetitive task, a good solution is to schedule it as a Cron job. The App Engine PaaS uses the App Engine Cron Service to trigger Cron jobs. For example, one might use a cron job to send out a daily digest email. In AppEngine a Cron job makes an HTTP GET request, at the configured scheduled date. The following code snippet configures the community detection Cron job.

```
<cronentries>
  <cron>
    <url>/tasks/detectCommunities</url>
    <description>nightly run the algorithm</description>
    <schedule>every day 00:00</schedule>
  </cron>
</cronentries>
```

The Cron jobs can be secured by preventing non-administrator users from accessing the URLs used by the scheduled tasks.

5.5 Task Queues

Often a request takes long to complete and it is not desirable to block the user interface and force the user to wait for a response. In this case, computing communities takes minutes but the user should be able to perform other tasks on the platform (read My Stream, post, comment, etc), while waiting for the result. Asynchronous requests do not wait for the API call to return a result. Google App Engine offers the asynchronous calls mechanism through the Task Queue API. If an application needs to execute work asynchronously (i.e. in the background), it adds this work as a task in the task queue. The tasks are then executed in a FIFO (First in, first out) order by the App Engine worker modules.

Chapter 6

Recommender Engine

We recall that the final goal of the project is to make recommendations based on the detected communities and the user's actions. Making recommendations based on inter-user comparison is called **collaborative recommendation** and it is one of the most well-known and used techniques in targeted advertising. The Recommender Engine is not a purely collaborative filtering system, but a Hybrid Recommender, where node attributes (i.e. user information) are combined with inter-user relations to make targeted recommendations. In Chapter 5 we have described how communities are detected inside Universe. In this chapter, we shall describe how this community structure contributes to making recommendations.

6.1 My Stream recommendations

The aim of My Stream is to deliver the right posts to the right people at the right time. Posts should be displayed in the order employees want to read them. A traditional approach is to display the posts in a reverse-chronological order. This is suitable for systems where the amount of new content posted every day is small. However, in Universe, more and more content is being produced every day. Being a platform for work, the time to absorb the new content has to be as short as possible. The employee should be presented with relevant content for his work and interests. A more user-centric approach, designed using the Recommender Engine, is to focus on what users should not miss. In a reverse-chronological approach, the user might not have time to go through all new posts and see only first posts. Rather than focusing on displaying all content in a chronological order, the Recommender Engine

tries to select what is more relevant to the user and display that first, leaving the one less relevant at the end. The higher the quality on the My Stream, the more involved employees are with the content (like, comment on posts). However, ranking posts according to their quality is not trivial. Various factors need to be taken into consideration, including having an up to date user profile.

There are two versions of My Stream in Universe. The first version is the traditional one and displays the posts in a reverse- chronological order. The second version is personalized in the sense that Universe retrieves the latest ten posts from the database. Next, it calls the Recommender Engine passing the retrieved list of posts. The Recommender Engine gets the user's community from the Graph Data Manager (deployed on the Compute Engine and with access to the Neo4j database). Then, it filters the list of posts based on the user who posted them. If the post has been shared by a user belonging to the current user's community, it is moved at the beginning of My Stream. In this way, the system takes care to display new content in a relevant order. The process repeats when the user scrolls down. The requests flow can be observed in Figure 6.1.

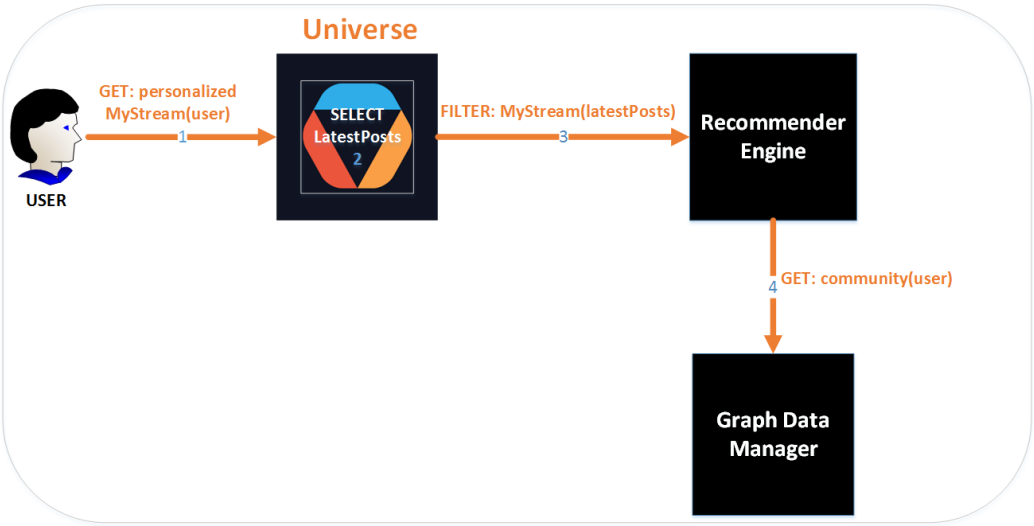

Figure 6.1: Personalized My Stream

6.2 My Workbench recommendations

The *My Workbench* section gives access to information such as today's meetings, their attached documents and participants, last edited Google Drive files and access links to them, the three most connected users, and the latest unread emails. The aim of this section is to gather useful information in one place, make the employee more productive and personalize the software experience. The information regarding Google Calendar Meetings, Google Drive Files and unread emails is retrieved using Google APIs (Calendar API, DriveAPI and GmailAPI)

6.3 Implementation

This section briefly presents the implementation details for the Recommender Engine. It tackles the approaches used for personalizing "My Stream" section and "My Workbench" section.

6.3.1 Personalized My Stream

To personalize "My Stream" based on the community structure, the following method is called from the ActivityEndpoint:

```
1 public List<Activity> getPersonalizedMyStream(User currentUser,
    List<Activity> chronologicalActivities, Integer page, Integer
     maxResults, Long[] excludedDiscussionIds)
```

The Activity object represents the post and among its fields it contains a content and an owner (the user who posted it). The methods iterates through the chronologicalActivities and adds the ones posted by the user's community at the beginning of the result list of Activity objects. Finally, it returns the filtered list to the user. In case no activities were posted by users in the community, the initial chronological list of activities is displayed.

6.3.2 My Workbench

To implement "My workbench", information from various sources needs to be combined. A Workbench model is composed of a list of DriveFile models of size three, a list of GmailEmail of size three, a list of String containing emails of the three most connected friends, a list of String containing the whole community of the user and a list of CalendarMeeting representing all meetings for the current day. The DriveFile model is defined by a file name and an access url. A CalendarMeeting is defined by an access url, a name, a list of participants and a list of DriveFile objects, representing the attachments. A GmailEmail is defined by a snippet (part of the email) and an access url. The community and the top three friends are retrieved from the Graph Data Manager, while the last accessed Drive files, calendar meetings and emails are retrieved from the corresponding Google APIs. The Google APIs used are:

1. Gmail API. [1]
2. Calendar API. [2]
3. Drive API. [3]

[1] https://developers.google.com/gmail/api/v1/reference/
[2] https://developers.google.com/google-apps/calendar/v3/reference/
[3] https://developers.google.com/drive/v3/reference/

Chapter 7

Evaluation

This section presents the results obtained after augmenting Universe with Universe Community Recommender.

7.1 Louvain and Neo4j Communities

Running the system with *Louvain* parameter, calls the Louvain implementation of the Community Detector. Figure 7.1 displays the communites obtained using Louvain clustering algorithm. The vertices have been saved in the Neo4j database. The visualization tool belongs to the Neo4j WebUI. The algorithm computed 9 communities for a company with 33 employees. The runtimes for the Louvain algorithm can be seen in Table 7.1.

7.2 Gephi Communities

If the system is run with *Gephi* parameter, it generates a file in a .gml format. The file contains the weighted adjacency matrix. After importing it into Gephi, Gephi's implementation of Louvain and Label Clustering algorithms can be run. All operations implemented in Gephi can be performed on it (visualization, layout application, formatted, filtered, etc). Figures 7.2 and 7.3 display the communities obtained by importing the gml file generated by Community Detector into Gephi 0.8.2. As it can be observed, for the company with 33 users, with Gephi's implementation of Louvain the number of communities decreased to 8. The one member communities correspond

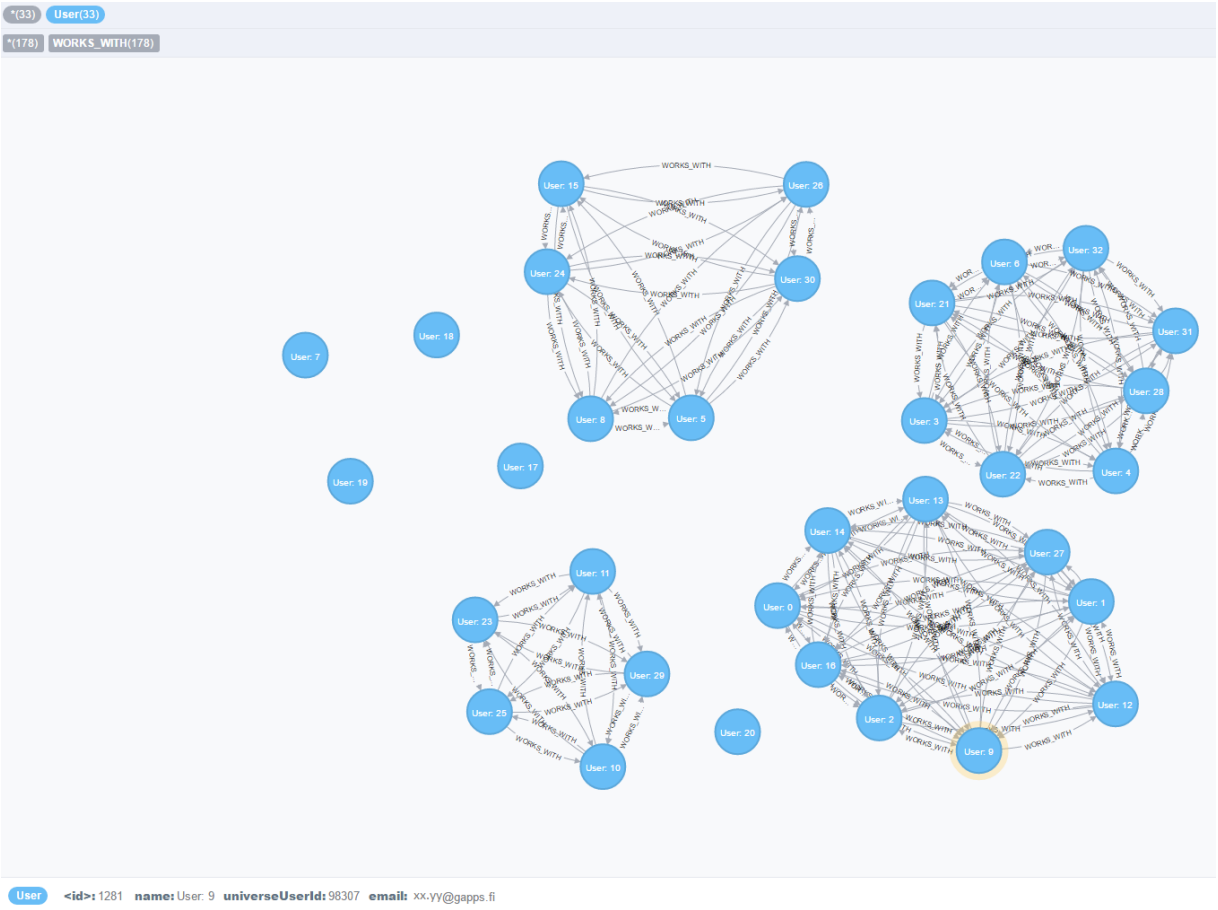

Figure 7.1: Company communities visualized with Neo4j. Algorithm used: Louvain

to the same users as in the previous case. The visualization tool used is the Gephi UI. As a comparison, the system also generates a .txt file which can be imported into cFinder. The communities obtained from CFinder (which supports overlapping communities) can be compared against the ones detected using Louvain, MCL or Gephi.

7.3 MCL and Neo4j Communities

Figure 7.4 displays the communities obtained by running the MCL alorithm. As it can be observed, the algorithm has the tendency of computing the strongest node and building a community around that node. In this case,

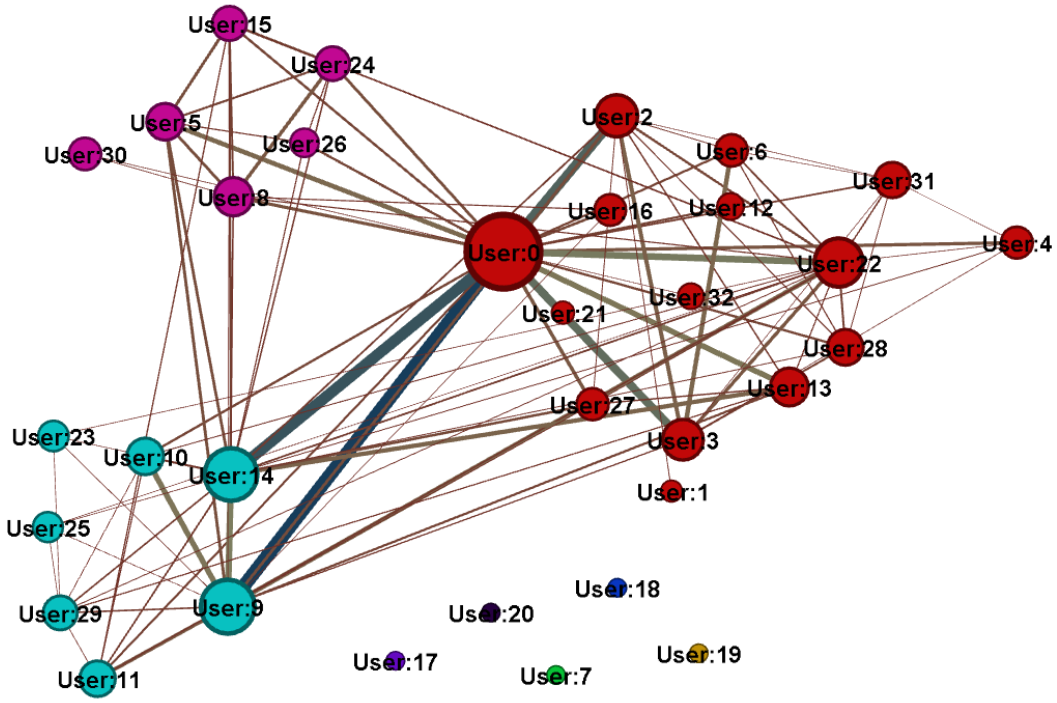

Figure 7.2: Company with 33 users. Communities visualized with Gephi

all the nodes except the ones that the previous algorithm had also mapped alone, are connected to the strongest node User 0 (having the largest weight to the other nodes).

7.4 Performance

Table 7.1 shows different running times for the main operations in the community detection phase. Computing the weights between users is the longest operation as it checks every pair of users ($O(n^2)$) for interactions. These interactions not only include data regarding channels or working departments, but it also makes requests to the Google APIs (Calendar API, Drive API) to check for calendar appointments or files. It is important to take into consideration that the channel weight computation is directly proportional with the total number of channels and posts in the company. Therefore, for a company that is intensively using Universe this operation will take longer. Moreover, the time to compute weights for the Google Drive and Google

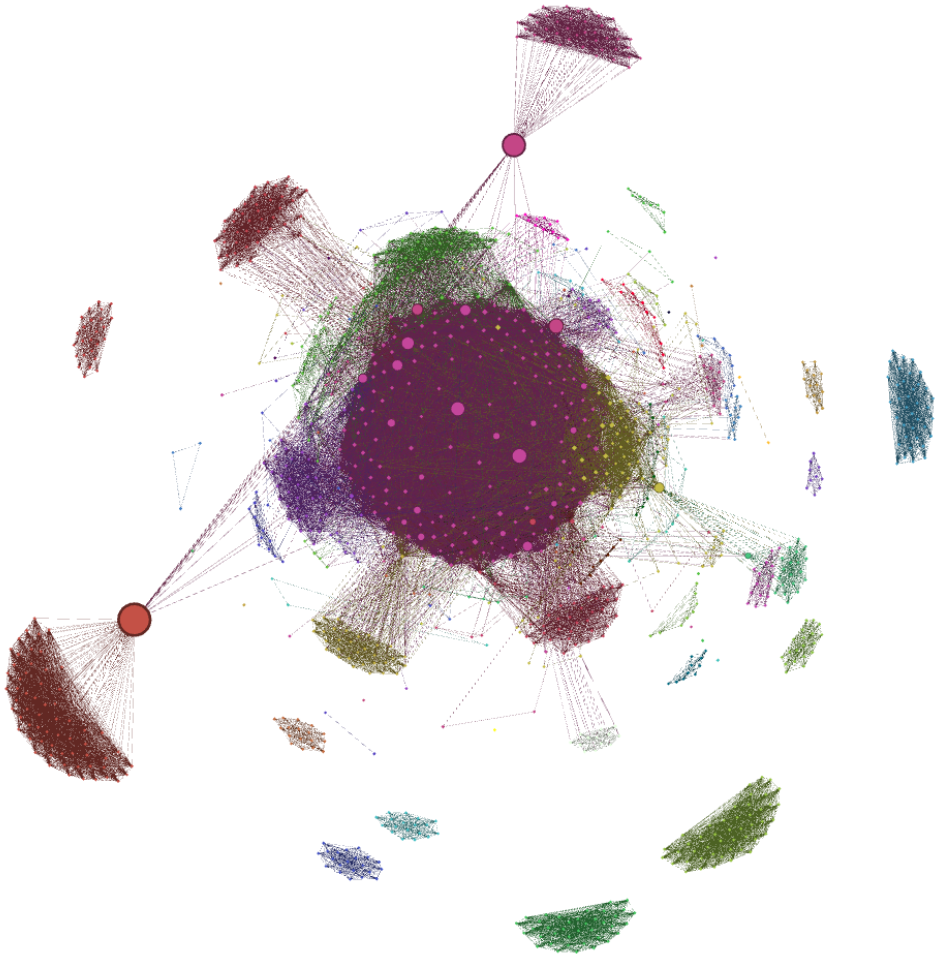

Figure 7.3: Company with 1113 users. Communities visualized with Gephi

Calendar is also directly proportional with the number of files and meetings the employees have.

One can also observe the running times for Louvain and MCL algorithms once the adjacency matrix is created. MCL algorithm proved to be faster than Louvain when the number of nodes is small, but Louvain algorithm scales very well when the number of users increases.

We then compare the time to store vertices to the Neo4j database and Google Datastore. As it can be observed, the time to persist data to the Google Datastore is significanly lower than persisting data to the Neo4j database. Hence, Google Datastore is more suitable for using in production, where the response time of the application is very important.

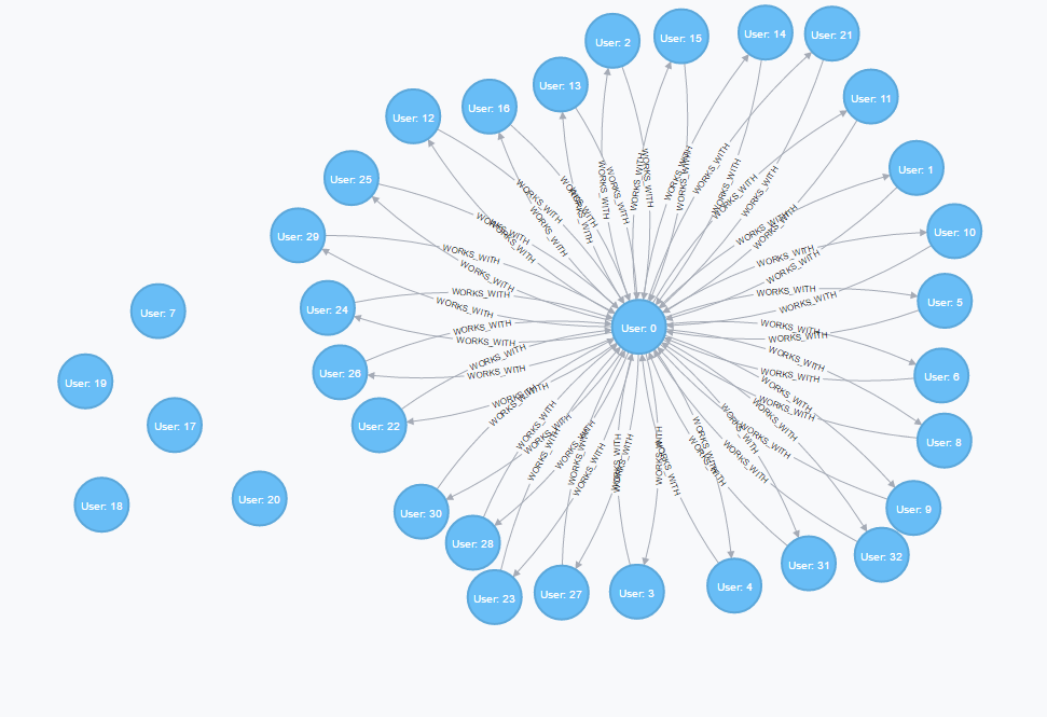

Figure 7.4: Company with 33 users. Communities visualized with Neo4j. Algorithms used: MCL

7.5 Recommendations

Firstly, "My Stream" is organized based on the communities the user is part of. The overhead for displaying posts depending on the community is approximately 100 ms. This comes from making the call to the Graph Manager for getting the community of the logged in user and rearranging the activities based on the user who posted them.

Secondly, each user receives suggested list of channels to join. These channels contain members from the user's community, hence they are suggested. The call to get suggested channels takes approximately 350 ms.

Finally, each user is presented with a "My Workbench" section with the user's unread emails, meetings for the current day, last edited Google Drive files and the user's community and top friends.

Figure 7.5 presents the results after one company's employees were asked whether the recommendations received on "My Workbench" were useful or

Table 7.1: Performance

No. users	Louvain Running Time (ms)	MCL Running Time (ms)	Computing weighted adjacency matrix (ms)	Persisting to Neo4j database (ms)	Persisting to Google Datastore (ms)
33	15	2	2300020 (with Google API calls) 4224	4800	50
296	21	457	528	9663	340
323	32	368	1078	10304	1655
1113	58	180622	3266	76795	2115

not useful. One employee voted that the recommendations were not useful, while the others voted useful.

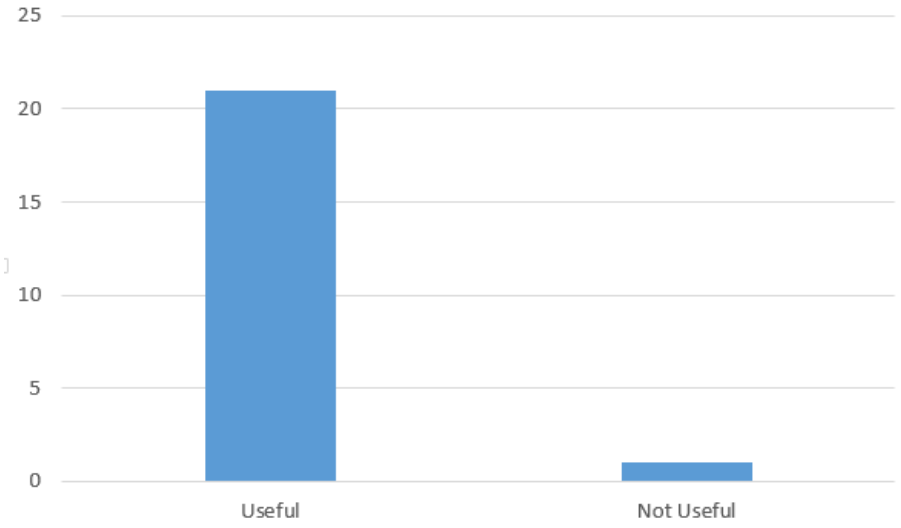

Figure 7.5: Users' voice

Chapter 8

Conclusions

This thesis presents Universe Community Recommender, a system integrating with Universe Intranet platform. Universe is used in companies to ease the communication and collaboration between employees and to gather all data related to the working environment in one place. Universe Community Recommender computes communities of users inside Universe and makes recommendations to users based on their actions and the communities they are part of. To build the graph of the company, the users have been mapped to vertices. To define the weights between users, a weight system that takes into consideration all possible interactions and collaborations between two users has been designed. The system constructs the weighted adjacency matrix, which is then used by one of the algorithms to cluster the vertices into communities. The vertices can be either stored in Neo4j or in Google Datastore, both scalable ACID-compliant transactional databases.

The selected algorithm for detecting communities is Louvain, a highly scalable, greedy approach. Louvain algorithm has been tested with up to 100 million nodes and detecting computing in a 2 million nodes network takes approximately 2 minutes on a standard PC [3]. In Universe, the number of users is expected to grow significantly. Hence, choosing a scalable method was one of the first criteria for selecting the algorithm. MCL algorithm was also tested with Universe data, but the number of detected communities was less than the ones obtained with Louvain method. When prompted with the communities obtained, the users rated the communities obtained with Louvain as more relevant. Moreover, the time to compute the communities with MCL method was higher due to the matrix multiplication.

Once the communities are detected, several recommendations are exposed

to the user. In the current implementation, the Universe user has access to his/her personalized "My Workbench" section, which gathers information from different sources such as Gmail API, Calendar API, Drive API, Community Detector. Moreover, the user is presented with a personalized "My Stream" section, where recent posts posted by his friends are displayed first on the page. Last but not least, the user receives a list of suggested channels that might be of interest. The HR managers also has access through the Neo4j Web UI to visualize communities, identify isolated users and improve teams. Different file types can be exported through the software, and then imported into other tools such as Gephi or CFinder for further explorations.

8.1 Future work

One aspect that could be considered as an improvement to the current system would be to take into consideration edge content when clustering vertices. For example, an edge can define social interaction between two users, while another edge can define work related interaction. Using this approach we could label the communities into working groups or social groups. This would provide more insight into the user data, but at the same time, the complexity of the community detection problem would be higher.

Furthermore, employees' skills could be used as a property when computing communities. This leads us to the first aspect, labelling communities. In an Intranet platform for work, it is important to classify people based on their skills and identify the right person for the right task as accurately as possible. Having communities of skills would simplify this process.

Moreover, feelings expressed in comments or posts are not taken into consideration. By default, the current implementation considers commenting on someone's post as a way of positive interaction, hence increasing weights between the two users. However, negative feelings could also be expressed through comments or posts. Detecting users' feelings is also known as affective computing or sentiment analysis. Other adjustments to the recommender engine include a gradual forgetting function as presented in [13] or setting weights directly proportional with the recency of inter-actions.

Next, detecting overlapping communities would be the most important improvement. It is often the case when a user is part of many communities at the same time. At a company level, a user could be part of the development

team, but also in charge of the management of this team, therefore part of two communities: development team and management team. Currently, the software exports a file that can be imported into CFinder, but this may be an arduous work to do for an HR manager, for example. Hence, the system should be extended by implementing its own overlapping community detection algorithm.

Last but not least, designing and implementing a personalized interactive visualization tool for the communities both for the HR administrator and the regular employee would bring a significant value to Universe. This would include advanced filtering criteria, zooming into communities or highlighting specific areas of the communities.

8.2 Extensibility

Universe Community Recommender could be used for other Intranet platforms by making some adequate adjustments. The VertexAdaptedService maps users to vertices by extracting relevant information about users. This can be easily reused as the information needed is an id and an email, fields that are present in all Intranet platforms for work. However, computing the weight between vertices has to be adapted according to the possible interactions among users. In Universe, users can be part of the same channel, comment and post, share Google Calendars and contribute to the same Google Drive file. Therefore, weights are computed based on these actions. However, these actions can be replaced according to the platform's needs.

The "My Workbench" section is thoroughly concise and only contains useful information for the users. Nonetheless, it can be extended with other information that another platform would consider useful for its users. For Universe, last three unread emails, last modified or viewed Drive file, calendar meetings for the current day and the user's most connected friends were considered as the most relevant information. The personalized "My Stream" and the automatic channel creation suggestions are rather specific to Universe. Other Intranet platforms might consider using the detected communities for other types of recommendations. Once the communities are detected, the software can be easily extended with other types recommendations.

All in all, Universe Community Recommender has been designed in a generic way, exposing a set of API methods to implement new clustering algorithms, but also some configuration classes, where for example, weights can be ad-

justed. However, the software respects the open/closed principle (open for extension, closed for modification), as the existing implementation including weights computation and recommendations generated for users (personalized My Stream, automatic channel creating suggestions) should not be changed, only extended or replaced.

Bibliography

[1] Barber, M. J., and Clark, J. W. Detecting network communities by propagating labels under constraints. *Phys. Rev. E 80* (August 2009), 026129.

[2] Barnes, E. R. An algorithm for partitioning the nodes of a graph. *Algebraic and Discrete Methods 4*, 3 (1982), 541–550.

[3] Blondel, V. D., Guillaume, J., Lambiotte, R., and Lefebvre, E. Fast unfolding of community hierarchies in large networks. *Physics reports, 2010 abs/0803.0476* (2008).

[4] Burke, R. D. Hybrid recommender systems: Survey and experiments. *User Model. User-Adapt. Interact. 12*, 4 (2002), 331–370.

[5] Chen, W., Liu, Z., Sun, X., and Wang, Y. Community detection in social networks through community formation games. In *IJCAI* (2011), T. Walsh, Ed., IJCAI/AAAI, pp. 2576–2581.

[6] Flake, G. W., Tarjan, R. E., and Tsioutsiouliklis, K. Graph clustering and minimum cut trees. *Internet Mathematics 1*, 4 (2003), 385–408.

[7] Fortunato, S. Community detection in graphs. *Physics Reports 486* (2010), 75–174.

[8] Fortunato, S., and Lancichinetti, A. Community detection algorithms: a comparative analysis: invited presentation, extended abstract. In *4th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '09, Pisa, Italy, October 20-22, 2009* (2009), G. Stea, J. Mairesse, and J. Mendes, Eds., ICST/ACM, p. 27.

[9] Gergely, P., Imre, D., Illes, F., and Tamas, V. Uncovering the overlapping community structure of complex networks in nature and society. *Nature 435* (2005), 814–818.

[10] Girvan, M., and Newman, M. E. J. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences 99*, 12 (2002), 7821–7826.

[11] Krishnamurthy, B., and Wang, J. On network-aware clustering of web clients. In *SIGCOMM* (2000), pp. 97–110.

[12] Kumpula, J. M., Kivelä, M., Kaski, K., and Saramäki, J. Sequential algorithm for fast clique percolation. *Phys. Rev. E 78* (August 2008), 026109.

[13] Montaner, M., López, B., and de la Rosa, J. L. A taxonomy of recommender agents on the Internet. *Artificial. Intelligence. Review 19*, 4 (2003), 285–330.

[14] Newman, M. E. J., and Girvan, M. Finding and evaluating community structure in networks. *Physical Review E 69*, 026113 (2004).

[15] Paul, A., and Choudhury, D. Community detection in social networks: An overview. *JRET: International Journal of Research in Engineering and Technology, 2013 02 Special Issue: 02*.

[16] Raghavan, U. N., Albert, R., and Kumara, S. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E 76* (April 2007), 036106.

[17] Reid, F., McDaid, A. F., and Hurley, N. J. Percolation computation in complex networks. In *International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2012, Istanbul, Turkey, 26-29 August 2012* (2012), IEEE Computer Society, pp. 274–281.

[18] Rodriguez, M. A., and Neubauer, P. Constructions from Dots and Lines. *Bulletin of American Society for Information Science & Technology September* (2010).

[19] Ronhovde, P., and Nussinov, Z. Multiresolution community detection for megascale networks by information-based replica correlations. *Phys. Rev. E 80* (July 2009), 016109.

[20] Rosvall, M., and Bergstrom, C. T. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences* (2008), 1118.

[21] van Dongen, S. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.

[22] Yang, J., and Leskovec, J. Overlapping community detection at scale: A nonnegative matrix factorization approach. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013* (2013), S. Leonardi, A. Panconesi, P. Ferragina, and A. Gionis, Eds., ACM, pp. 587–596.

[23] Yang, J., McAuley, J. J., and Leskovec, J. Community detection in networks with node attributes. In *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013* (2013), pp. 1151–1156.